AD-A061 932 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
LABORATORY FOR COMPUTER SCIENCE (FORMERLY PROJECT MAC) PROGRESS--ETC(U)
OCT 78 M L DERTOUZOS N00014-75-C-0661

UNCLASSIFIED LCS-PR-14 NL

1 OF 2
AD A061932


932

LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Progress Report XIV

JANUARY - DECEMBER 1976

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: LCS Progress Report XIV	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Laboratory for Computer Science (formerly Project MAC) Progress Report XIV January-December 1976 AD61246	5. TYPE OF REPORT & PERIOD COVERED: ARPA-DOD Progress Report 1/76-12/76
	6. PERFORMING ORG. REPORT NUMBER: LCS/PR-XIV
7. AUTHOR(s): Laboratory for Computer Science Participants M.L. Dertouzous, Director	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: LABORATORY FOR COMPUTER SCIENCE (formerly Project MAC) Massachusetts Institute of Technology 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency Department of Defense 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE: Oct. 16, 1978
	13. NUMBER OF PAGES: 156
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research Department of the Navy Information Systems Program Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Real-time Computers	Computer Languages	Automata Theory
On-line Computers	Computer Networks	Morse-Code
Multi-access Computers	Information Systems	Knowledge-Based Systems
Dynamic Modelling	Programming Languages	Complexity
Computer Systems	Computation Structures	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Annual Summary Report of progress made at the Laboratory for Computer Science under this contract during the period January-December 1976.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 12 1 007

Work reported herein was carried out within the Laboratory for Computer Science (formerly Project MAC), a Massachusetts Institute of Technology interdepartmental laboratory. Support was provided by the Advanced Research Projects Agency of the Department of Defense, under Office of Naval Research Contract N00014-75-C-0661.

(6) LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)
PROGRESS REPORT XIV

JANUARY - DECEMBER 1976

(14) LCS-PR-14

(9) Annual progress rept. 1 Jan-31 Dec 76,

(10) M. L. /Dertouzos

(15) N00014-75-C-0661

LABORATORY FOR COMPUTER SCIENCE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

(11) 16 Oct 78

(12) 145 p.

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION	1
LABORATORY FOR COMPUTER SCIENCE ADMINISTRATION	3
COMPUTER SYSTEMS RESEARCH GROUP	5
A. Introduction	7
B. The Information Sharing Kernal Design Project	7
C. Research Problems of Decentralized Systems With Largely Autonomous Nodes	9
D. A Local Network for LCS	16
E. ARPANET and NSW Support	19
DOMAIN SPECIFIC SYSTEMS RESEARCH GROUP	25
A. Introduction	27
B. Real Time Block Diagram Schemata	27
C. Semantics for Distributed Processing	28
D. Automatic Code Generation	29
E. Process Control	30
F. Microprocessor Simulation of Digital Logic	31
G. Laboratory	33
KNOWLEDGE-BASED SYSTEMS GROUP	37
A. Summary of Work in Progress	39
PROGRAMMING METHODOLOGY GROUP	41
A. Introduction	43
B. Iterators	45
C. Access Control	47
D. Optimization	55
E. Specifications for Data Abstractions	61
PROGRAMMING TECHNOLOGY GROUP	73
A. Introduction	75
B. Morse Code	75
C. Interpersonal Communication	89
D. Other Projects	107
LABORATORY FOR COMPUTER SCIENCE PUBLICATIONS	115

INTRODUCTION

This annual progress report to the Advanced Research Projects Agency (ARPA) of the Department of Defense describes research performed at the M.I.T. Laboratory for Computer Science (formerly Project MAC), funded by that agency and monitored by the Office of Naval Research during the period January 1-December 31, 1976.

The Laboratory was organized at M.I.T. in 1963 to conduct research in Time-Shared Computer Systems. Contributions of L.C.S. include the Compatible Time-Sharing System (CTSS), Multics, the mathematical-expert program MACSYMA, and a variety of programming languages, systems and techniques. The research described in this report reflects the current research directions of the Laboratory, oriented to promising areas as well as pressing technological needs of the computer science field.

During the reporting period (January 1976-December 1976), L.C.S. personnel numbered approximately 251 people, including 34 faculty, 61 research and support staff members, 101 graduate students, 50 undergraduate students, and 5 visiting researchers and scientists.

The main focus of the research reported herein has been in the reduction of the substantive and increasing costs associated with the generation, maintenance and documentation of programs. In particular, work carried out by the Knowledge-Based Systems group focused on the identification of a very high level language in which inventory control programs are specified, and on the associated compiler that translates such a program to PL/1 code. In the Domain Specific Systems Research group research commenced on the programming of microcomputers from high-level languages for such purposes as the automatic control of physical processes, maintenance and instrumentation.

The Programming Technology group concentrated its research on the development of a Morse Code system. Through this system the group seeks to understand and develop techniques for embedding a great deal of structural knowledge (in this case about Morse Code) into computer programs.

The Computer Systems Research group focused its research on the analysis and certification of large systems using the MULTICS systems as its principal model and laboratory. In addition, work was inflated on a local-network that will link the laboratory's computational resources. The Programming Methodology group continued the development of the structured programming language CLU which has a modular construction that facilitates the representation of abstractions.

Acknowledgements

Assembly and compilation of this report was done by Paulyn Heinmiller. Illustrations were done by Allison Platt. Amended illustrations by Sara Geitz.

ADMINISTRATION

Academic Staff

M. L. Dertouzos	Director
J. Moses	Associate Director

Administrative Staff

M. E. Baker	Administrative Assistant
L. G. Daniels	Librarian
H. S. Hughes	Administrative Services
C. P. Kent	Assistant Fiscal Officer
T. L. Lightburn	Fiscal Consultant
G. W. Oro	Fiscal Officer
A. A. Platt	Information Services
D. C. Scanlon	Administrative Officer
G. L. Wallace	Purchasing Agent

Support Staff

G. W. Brown
L. S. Cavallaro
M. Cummings
P. G. Heinmiller
J. Jones
D. Kontrimus
M. K. Martucci
E. M. Profirio
T. Sealy
R. Varjebedian
L. Withers

COMPUTER SYSTEMS RESEARCH

Academic Staff

J. H. Saltzer, Group Leader
D. D. Clark
F. J. Corbató
D. D. Redell
M. D. Schroeder
L. Svobodova

Research Staff

N. C. Federman
R. J. Kanodia
R. F. Mabee
K. T. Pogran
D. M. Wells

Graduate Students

A. J. Benjamin
E. C. Ciccarelli
H. C. Forsdick
R. M. Frankston
H. J. Goldberg
A. R. Huber
D. H. Hunt
P. A. Janson
P. A. Karger
S. T. Kent
A. W. Luniewski
A. H. Mason
W. A. Montgomery
R. P. Reed
M. Shibuya
K. R. Sollins

Undergraduate Students

C. R. Davis
C. R. D'Oliveira
E. S. Harriman
R. P. Planalp
H. Rodriguez, Jr.
S. A. Swernofsky

Support Staff

V. M. Newcomb
M. F. Webber

COMPUTER SYSTEMS RESEARCH

A. INTRODUCTION

During this year, the Computer Systems Research group completed one major project, the information sharing kernel design project, and made significant progress on two others, the study of distributed systems and implementation of a local network. We also continued support of the ARPANET and NSW on Multics. These activities are described in the following sections.

B. THE INFORMATION SHARING KERNEL DESIGN PROJECT

This year we completed a three year project to carry out engineering studies whose goal was to demonstrate the feasibility of producing a full function general purpose operating system whose central supervisor code is simple enough that its correct operation can be certified by some form of auditing. The term "security kernel" is often used to describe this body of critical code, since the functions that must be included in this code are precisely those that insure the correct operation of the system, and insure the integrity of the information stored in the system. This engineering study was part of a larger project, the Guardian project, to produce a prototype of a certifiable operating system, based on the Multics system. The Guardian project included development of models to characterize security in a computer system, development of formal specification techniques for operating systems, and actual implementation of a system matching the models.

The general strategy of this engineering study involved identifying all reasonable sounding proposals for simplifying the Multics kernel, and selecting for trial implementation those that could not be accepted as obviously straightforward or rejected as obviously inappropriate. Three kinds of redesign proposals emerged:

a. Removing from the kernel those formerly protected supervisor functions that did not really require that protection

b. Taking advantage whenever possible, of the natural separation afforded by processes in distinct address spaces communicating at arm's length to implement protection functions

c. Using more systematic program structuring techniques for implementing the remaining kernel functions, so that the result might be easier to verify.

Probably the most interesting and important result of this work is the invention of a file system and processor multiplexing organization that eliminates the complicating cycles of dependency normally found among the modules of an operating system kernel. The organization is based on the discipline of type extension, a strategy that has been

used previously to organize application programs, but has heretofore not been applied to the structure of an operating system itself. Inside an operating system, careful analysis is required to identify all intermodule dependencies. The opportunity exists, for example, for an operating system module to produce dependency loops by participating in the implementation of its own execution environment. Such opportunities are less of a problem for application programs, which typically depend on the operating system to provide their execution environment. Our study suggests that in a properly structured system, all dependencies that cannot be eliminated will fall into one of five categories, as follows. A module M is dependent on some other module if and only if:

a. The other module manages some object that is a component of the object defined by M

b. That module provides a map used to relate names used by M to lower level objects

c. That module provides the containers for the algorithms and temporary storage for M

d. That module defines the address space in which M executes

e. That module implements the interpreter (the real or virtual processor) that executes the algorithms of M.

Using the rationale just described, and with the five kinds of dependencies in mind, it was possible to design a loop-free structure of object managers that implement the complete functionality required in the Multics kernel.

We summarize our experience in applying the type extension rationale to structuring the Multics kernel as follows. Most systems appear to have a loop-free dependency structure if viewed from far enough away. The obvious component relationships and the obvious operations follow loop-free paths among the modules. On close inspection, however, map, program, address space, and interpreter dependencies will almost certainly generate loops in the system designed without loop avoidance as a primary objective. The map, program and address space loops usually are easily broken (at least during the design stage) by introducing new object types to store the maps, programs, and address space definitions. The interpreter dependency loops appear to be eliminated in most systems by using a two level implementation of processes. The most difficult and subtle structural problems are caused by exception handling--especially when the exceptions are part of the mechanisms that control resource usage. The difficulty is partly intrinsic--such exceptions tend to occur at low levels in the system but are related to high level objects--and partly methodological--resource usage controls and the paths followed to deal with exceptions tend to be added to a design last.

It was our expectation that the structural simplifications to the kernel would be accompanied by a reduction in the size of the kernel, as measured in lines of source code. The size of the Multics kernel at the start of the project was 54,000 lines of source code, a bulk sufficiently staggering to inhibit any serious thought of conclusive auditing. Our application of the three design procedures mentioned above produced a version of the kernel approximately half the size of the original. We expect further size reductions would be possible were our proposals carried through to all areas of the kernel to which they would apply. An unresolved question is whether the kernel must enforce all security requirements, or only those related to some external standard such as the military model of non-discretionary levels and categories. Had our kernel enforced only the latter, it would have been somewhat smaller, though considerable work seems necessary to decide exactly how much smaller.

Experiments with components of the system that we rewrote indicate that the structural modifications we proposed did not have a significant performance impact on the system, and we conclude that a secure system need have no performance penalty. The most serious impact on performance in our work comes from the use of a high level language, and presumably this difficulty could be minimized if a high level language were used that is easier to compile efficiently than full PL/I.

The primary conclusion of this project is that the kernel of a general purpose operating system can be made significantly simpler by first imposing clear criteria as to what should be in it--the kernel concept--and second, a design discipline based on type extension. It is also apparent that minor adjustments of the underlying hardware architecture can make a significant difference in operating system complexity, and similarly that minor variations in the semantics of the user interface can make major differences in the complexity of implementation of the kernel.

C. RESEARCH PROBLEMS OF DECENTRALIZED SYSTEMS WITH LARGELY AUTONOMOUS NODES

A currently popular systems research project is to explore the possibilities and problems for computer system organization that arise from the rapidly falling cost of computing hardware. Interconnecting fleets of mini- or micro-computers and putting intelligence in terminals and concentrators to produce so-called "distributed systems" has recently become a booming development activity. While these efforts range from ingenious to misguided, many seem to miss a most important aspect of the revolution in hardware costs: that more than any other factor, the entry cost of acquiring and operating a free-standing, complete computer system has dropped and continues to drop rapidly. Where a decade ago the capital outlay required to install a computer system ranged from $150,000 up into the millions, today the low end of that range is below $15,000 and dropping.

The consequence of this particular observation for system structure comes from the next level of analysis. In most organizations, decisions to make capital acquisitions tend to be more centralized for larger capital amounts, and less centralized for smaller capital amounts. On this basis we may conjecture that lower entry costs for computer systems will lead naturally to computer acquisition decisions being made at lower points in a management hierarchy. Further, because a lower level organization usually has a smaller mission, those smaller priced computers will tend to span a smaller range of applications, and in the limit of the argument will be dedicated to a single application. Finally, the organizational units that acquire these computers will by nature tend to operate somewhat independently and autonomously from one another, each following its own mission. From another viewpoint, administrative autonomy is really the driving force that leads to acquisition of a computer system that spans a smaller application range. According to this view, the large multiuser computer center is really an artifact of high entry cost, and does not represent the "natural" way for an organization to do its computing.

A problem with this somewhat oversimplified analysis is that these conjectured autonomous, decentralized computer systems will need to communicate with one another. For example: the production department's output will be the inventory control department's input, and computer-generated reports of both departments must be submitted to higher management for computer analysis and exception display. Thus we can anticipate that the autonomous computer systems must be at least loosely coupled into a cooperating confederacy that represents the corporate information system. This scenario describes the corporate computing environment, but a similar scenario can be conjectured for the academic, government, military, or any other computing environment.

The key consequence of this line of reasoning for computer system structure, then, is a technical problem: to provide coherence in communication among what will inevitably be administratively autonomous nodes of a computer network. Technically, autonomy appears as a force producing incoherence: one must assume that operating schedules, loading policy, level of concern for security, availability, and reliability, update level of hardware and software, and even choice of hardware and software systems will tend to vary from node to node with a minimum of central control. Further, individual nodes may for various reasons occasionally completely disconnect themselves from the confederacy, and operate in isolation for a while before reconnecting. Yet to the extent that agreement and cooperation are beneficial, there will be a need for communication of signals, exchange of data, mutual assistance agreements, and a wide variety of other internode interaction. We hypothesize that one-at-a-time ad hoc arrangements will be inadequate, because of their potentially large number and the programming cost in dealing with each node on a different basis.

Coherence can be sought in many forms. At one extreme, one might set a company-wide standard for the electrical levels used to drive point-to-point communication lines that interconnect nodes or that attach any node to a local communication network. At the opposite extreme, one might develop a data management protocol that allows any user of any node to believe that there is a central, unified database management system with no identifiable boundaries. The first extreme might be described as a very low-level protocol, the second extreme as a very high-level protocol, and there seem to be many levels in between, not all strictly ordered.

By now, considerable experience has been gained in devising and using relatively low-level protocols, up to the point that one has an uninterpreted stream of bits flowing from one node of a network to another. The ARPANET and TELENET are perhaps the best-developed examples of protocols at this level, and local networks such as the ETHERNET and the Irvine Ring network provide a similar level of protocol on a geographically smaller scale. In each of those networks, standard protocols allow any two autonomous nodes (of possibly different design) to set up a data stream from one to the other; each node need implement only one protocol, no matter how many other differently designed nodes are attached to the network. However, standardized coherence stops there; generally each pair of communicating nodes must make some (typically ad hoc) arrangement as to the interpretation of the stream of bits: does it represent a stream of data, a set of instructions, a message to one individual, etc. For several special cases, such as exchange of mail or remotely submitting batch jobs, there have been developed higher-level protocols; there tends to be a distinct ad hoc higher-level protocol invented for each application. A Master's thesis by Paul Levine explored some of the problems of protocols that interpret and translate data across machines of different origin.

The image of a loose confederacy of cooperating autonomous nodes requires at a minimum the level of coherence provided by these networks; it is not yet clear how much more is appropriate, only that the opposite extreme in which the physically separate nodes effectively lose their separate identity is excluded by the earlier arguments for autonomy. Between lies a broad range of possibilities that need to be explored.

1. Coherence and the Object Model

During the current year, members of the Computer Systems Research group held a graduate-level seminar that explored this area of coherence among interconnected systems, and developed a framework for discussion that allows one to pose much more specific questions. The first conclusion of this work is that to put some structure on the range of possibilities, it is appropriate to think first in terms of familiar semantic models of computation, and then to inquire how the semantic model of the behavior of a single node might be usefully extended to account for interaction with other, autonomous nodes. To get a concrete starting point that is as developed as possible, we gave initial

consideration to the object model. (Two other obvious candidates for starting points are the data flow model and the actor model, both of which already contain the notion of communications; since neither is developed quite as far as the object model we have left them for future examination.) Under that view, each node is a self-contained system with storage, a program interpreter that is programmed in a high-level object-oriented language such as CLU or Alphard, and an attachment to a data communication network of the kind previously discussed.

We immediately observed that several interesting problems are posed by the interaction between the object model and the hypothesis of autonomy. There are two basic alternative premises that one can start with in thinking about how to compute with an object that is represented at another node; send instructions about what to do with the object to the place it is stored, or send a copy of the representation of the object to the place that wants to compute with it. (In between combinations are also possible, but conceptually it is simpler to think about the extreme cases first.) An initial reaction might be to begin by considering the number of bits that must be moved from one node to another to carry out the two alternatives, but that approach misses the most interesting issues: reliability, integrity, responsibility for protection of the object, and naming problems. Suppose the object stays in its original home. Semantics for requesting operations, and reporting results and failures are needed. For some kinds of objects, there may be operations that return references to other, related objects. Semantics to properly interpret these references are required. Checking of authorization to request operations is required. Some way must be found for the (autonomous) node to gracefully defer, queue, or refuse requests, if it is overloaded or not in operation at the moment.

Suppose, on the other hand, that a copy of the object is moved to the node that wants to do the computation. Privacy, protection of the contents, integrity of the representation, and proper interpretation of names embedded in the object representation are all problems. Yet, making copies of data seems an essential part of achieving autonomy from nodes that contain needed information but aren't always accessible. Considering these two premises as alternatives seems to raise simultaneously so many issues of performance, integrity of the object representation, privacy of its content, what name is used for the object, and responsibility for the object, that the question is probably not posed properly. However, it begins to illustrate the range of considerations that should be thought about. We have identified the following, more specific, problems that require solutions:

a. One would expect to achieve reliability and response speed by arranging that an object have multiple representations stored at different places. However, such replication must be done in a systematic way. An example of non-systematic multiple representation occurs whenever one user of a time-sharing system confronts another with the complaint, "I thought you said you fixed that bug," and receives the response, "I did. You must have gotten an old copy of the program.

What you have to do is type..." Semantics are needed to express the notion that for some purposes any of several representations are equally good, but for other purposes they aren't.

b. An object at one node needs to "contain" (for example, use as part of its representation) objects from other nodes. This idea focuses on the semantics of naming remote objects. It is not clear whether the names involved should be relatively high-level (e.g., character-string file names) or low-level (e.g., segment numbers).

c. Related to the previous problem are issues of object motion: suppose object A, which contains as a component object B, is either copied or moved from one node to another, either temporarily or permanently. Can object B be left behind or be in yet another node? The answer may depend on the exact combination of the attributes: copy or moved, temporary or permanent. Autonomy is deeply involved here, since one cannot rely on availability of the original node to resolve the name of B.

d. More generally, semantics are needed for gracefully coping with objects that aren't there when they are requested. (Information stored in autonomous nodes will often fall in this category.) This idea seems closely related to the one of coping with objects that have multiple versions and the most recent version is inaccessible. (Semantics for dealing systematically with errors and other surprises have not really been devised for monolithic, centralized systems either. However, it appears that in the decentralized case, the problem cannot so easily be avoided by the ad hoc tricks or finesse as it was in the past.)

e. Algorithms are needed that allow atomic update of two (or more) objects stored at different nodes, in the face of errors in communication and failures of individual nodes. (Most published work on making atomic updates to several sites has concentrated on algorithms that perform well despite communication delay or that can be proven correct. Unfortunately, algorithms constructed without consideration of reliability and failure are not easily extended to cope with those additional considerations, so there seems to be no way to build in that work.) There are several forms of atomic update: there may be consistency constraints across two or more different objects (e.g., the sum of all the balances in a bank should always be zero) or there may be a requirement that several copies of an object be kept identical. The semantic view that objects are immutable may provide a more hospitable base for extension to interaction among autonomous nodes than the view that objects ultimately are implemented by cells that can contain different values at different times. (The more interesting algorithms for making coordinated changes in the face of errors seem to implement something resembling immutable objects.)

Constraining the range of errors that must be tolerated seems to be a promising way to look at these last two problems. Not all failures are equally likely, and more important, some kinds of failures can perhaps be guarded against by specific remedies, rather than tolerated. For example, a common protocol problem in a network is that some node both crashes and restores service again before anyone notices; outstanding connections through the network sometime continue without realizing that the node's state has been reset. A change in the semantics of the host-net interface could locally eliminate this kind of failure instead of leaving it as a problem for higher level protocols.

The following oversimplified world view, to be taken by each node, may offer a systematic way to think about multiply represented objects and atomic operations: there are two kinds of objects, mine and everyone else's. My node acts as a cache memory for objects belonging to others that I use, and everyone else acts as a backing store. These roles are simply reversed for my own objects. (One can quickly invent situations where this view breaks down, causing deadlocks or wrong answers, but the question is whether or not there are real world problems for which this view is adequate.)

Finally, it is apparent that one can get carried away with ingenious algorithms that handle all possible cases. An area requiring substantial investigation is real world applications. It may turn out that only a few of these issues arise often enough in practice to require systematic solutions. It may be possible, in many cases, to cope with distant objects quite successfully as special cases to be programmed one at a time.

2. Other Problems in the Semantics of Coherence

Usual models of computation permit only "correct" results, with no provision for tolerating "acceptably close" answers. Sometimes provision is made to report that no result can be returned. In a loose confederacy of autonomous nodes, exactly correct results may be unattainable, but no answer at all is too restricting. For example, one might want a count of the current number of employees, and each department has that number stored in its computer. At the moment the question is asked, one department's computer is down, and its count is inaccessible. But a copy of last month's count for that department is available elsewhere. An "almost right" answer utilizing last month's count for one department may well be close enough for the purpose the question was asked, but we have no semantics available for requesting or returning such answers. A more extreme example surrounds an attempt to determine the sum of all checking account balances in the United States, by interrogating every bank's computer. An exact result seems both unnecessary and unrealistic to obtain. A general solution to this problem seems to require a perspective from Artificial Intelligence, but particular solutions may be programmable if there were available semantics for detecting that one object is an out-of-date version of another, or that a requested but unavailable object has an out-of-date copy. It is not clear at what level these associations should be made.

Semantics are also needed to express constraints or partial contraints of time sequence. (e.g. "reservations are to be made in the order they are requested, except that two reservation requests arriving at different nodes within one minute may be processed out of order.") Note that the possibility of unreliable nodes or communications severely complicates this problem.

The semantics of autonomy are not clear. When can I disconnect my node from the network without disrupting my (or other) operations? How do I refuse to report information that I have in my node in a way that is not disruptive? If my node is overloaded, which requests coming from other nodes can be deferred without causing deadlock?

3. Heterogeneous and Homogeneous Systems

A question that we have repeatedly encountered is whether or not one should assume that the various autonomous nodes of a loosely coupled confederacy of systems are identical either in hardware or in lower level software support. The assumption of autonomy and observations of the way the real world behaves both lead to a strong conclusion that one must be able to interconnect heterogeneous (that is, different) systems. Yet, to be systematic, some level of homogeneity is essential, and in addition the clarity that homogeneity provides in allowing one to see a single research problem at a time is very appealing.

We now believe that the proper approach to this issue lies in careful definition of node boundaries. We insist that every node present to every other node a common, homogeneous interface, whose definition we hope to specify. That interface may be a native interface, directly implemented by the node, or it may be simulated by interpretation, using the (presumably different) native facilities of the node. This approach allows one to work on the semantics of decentralized systems without the confusion of heterogeneity, yet it permits at least some non-conforming systems to participate in a confederacy. There is, of course, no guarantee that an arbitrary previously existing computer system will be able to simulate the required interface easily or efficiently.

4. Conclusion

The various problems uncovered in the course of this work are by no means independent of one another, although each seems to have a flavor of its own. In addition, they probably do not span the complete range of issues that should be explored in establishing an appropriate semantics for expressing computations in a confederacy of loosely coupled, autonomous computer systems. Further, some are recognizable as problems of semantics of centralized systems that were never solved very well. But they do seem to represent a starting point that we expect to lead to more carefully framed questions and eventually some new conceptual insight.

D. A LOCAL NETWORK FOR LCS

During the year, development of the Local Network for the Laboratory for Computer Science progressed to the point where the first three nodes on the net are expected to be operational within the next two months. As discussed in detail in the sections below, the critical decisions concerning the hardware and protocols to be used on our network have been made during the last twelve months, making it possible for a variety of projects related to the network to proceed forward in parallel.

1. Hardware

Our last annual report related that our choices for the transmission technology to be used in the network quickly narrowed to two architectures: the ethernet developed by Boggs and Metcalfe at Xerox Palo Alto Research Center, and the ring network developed by Farber at the University of California, Irvine. The architecture and hardware of the ring network and the ethernet are very different, and, at first glance, the functional capabilities of the two seem quite different as well. However, discussions with Metcalfe and Farber, and with others in our laboratory, led to the conclusion that there are few inherent differences in the functional capabilities of the basic ethernet and ring network communications schemes. This made the choice between them a very difficult one. It appeared, in fact, that the important differences between the two networks were operational differences such as reliability, cost, and convenience, which could only be evaluated by comparing a running version of each network in a similar environment.

A way out of this dilemma was suggested when we discovered that we could design a network interface that, with minor modification, could operate either a ringnet or an ethernet. Thus, without procuring two complete sets of interface hardware, we can bring up both versions of the network and compare them operationally. Given this observation, we determined that we would construct the LCS Net in two subcomponents, one a ringnet and one an ethernet, and perform an operational comparison of the two. We have done some preliminary comparative analysis of the two.

The primary hardware component of our network is the Local Net Interface (LNI), which provides the means of connecting the various hosts to the network. The LNIs that we intend to use for the network have been designed by David Farber at the University of California, Irvine; they are a second generation ring interface that Farber is developing under contract with ARPA, based on the ring developed for the Irvine Distributed Computing System. We have been assisting in the design of these interfaces, so that we will be able to produce a version of this hardware that can drive an ethernet as well as a ringnet.

The LNI, as delivered by Farber, includes an interface to the PDP/11 Unibus. One of the tasks yet to be completed is the fabrication of an interface to connect the LNI to the PDP-10s in the building. It is possible that Farber will complete the design of a PDP-10 interface to the LNI; as an interim interface it appears very easy to attach the LNI to the TTL bus that is locally used for connection to the PDP-10s. Eventually, the LNI will probably require a connection to the PDP-10s that runs at a higher speed than the TTL bus will permit.

A hardware project that was partially completed during the year is the interconnection of a microprocessor to the LNI. A microprocessor directly connectable to the network can be used in a variety of ways, for example as a controller for a computer terminal or other remote input/output device. The microprocessor selected for this first implementation was the Motorola M6800. The first application for the microprocessor will be as a terminal interface for the local network.

One of the important functions of our local network will be to provide a means of access to the ARPANET from the various machines at the laboratory. The interconnection between the local net and the ARPANET will be made using a PDP 11/35 that was provided for the project by ARPA. This machine will be used to perform the various protocol translations that will be required as part of the interconnection of the local network and the ARPA network. One project being performed at the laboratory is the development of a hardware interface to connect this PDP/11 to the ARPANET. The DEC interface is bulky, expensive, and not rapidly obtainable. We hope our local version will perform better on these counts.

2. Protocols

As part of the development of our local network, it was necessary for us to develop or select a low level protocol for end-to-end communication over the network. We chose as a starting point the Transmission Control Protocol, or TCP, but we permitted ourselves the option of changing the protocol slightly to better conform to our local needs as we saw them. The resulting protocol is called Data Stream Protocol, or DSP. DSP provides functionality equivalent to TCP, but is simpler, primarily due to the elimination of certain control functions and synchronizing algorithms.

We are currently involved in an effort to bring DSP and TCP together again, since TCP is the ARPANET standard for end-to-end communication in the "internet" environment. We have attended several meetings of the TCP working group, and have met with some success in our attempt to include in TCP some of the features in DSP.

DSP must be implemented on all the machines which we propose to connect to the local network. Our initial effort has been devoted to an implementation of DSP for the UNIX operating system on the PDP/11. One of the first machines to be connected to our local network will be the UNIX system in the Domain Specific Systems research group. In

addition, the PDP/11 gateway to the ARPANET will run the UNIX operating system. An implementation of DSP (or perhaps TCP) is scheduled for the Multics system later in the calendar year. Preliminary plans have been made for implementation of DSP on the ITS machines, and we are considering how DSP might be implemented on the TENEX operating system. As part of the microprocessor project mentioned above, we have also implemented DSP for the M6800. The initial implementation on the M6800 required 1300 bytes of program, and although this size will undoubtedly increase as the implementation is polished, the size of the algorithm suggests that we were somewhat successful in our ambition that DSP be a fairly simple protocol.

Initially, the local net will use the same high level protocols that are now used in the ARPANET. It appears that the ARPANET protocols for remote login (TELNET), file transfer, and mail sending can be made to operate on top of DSP without major modification. Therefore, for systems that currently have software for connection to the ARPANET, the only coding required as part of the interconnection to the local net is the implementation of DSP, and minor modification of existing higher level protocols. ARPANET software already exists for all the machines currently scheduled for connection to the local network.

We have begun the design of higher level protocols to provide new services that seem appropriate in the local net. In particular, we have proposed a rather flexible scheme for naming and initiating connections to services in the local network. Examples of services that might be named using this mechanism are the delivery of a message to a specified mailbox, the updating of a file, or the remote login to a system. The mechanism uses decentralized active agents to provide an environment that is robust in the face of system failures. The names used are tree structured in order to deal in the natural way with name conflicts and to allow the easy definition of new services in a given context.

All of the network architectures that we have considered are completely insecure, since all messages being sent appear on all portions of the network. While our laboratory is a "benign" environment in which the needs for security of data communication are rather small, considerations of personal privacy continue to be relevant in an environment such as ours, so our needs for security, while minimal, are not zero. Also, we would like to design a network whose applicability extends to situations with stronger protection requirements than we have. For these reasons, we have studied the securing of information flowing through our local network by means of data encryption. Data encryption is becoming a viable possibility for a network even as simple as the one we contemplate here, because data encryption algorithms can now be obtained on a single chip. We have proposed an end-to-end encryption strategy using the NBS data encryption standard integrated into a modified version of DSP, which is essentially invisible to the higher level protocols. Its use in the local network could be made automatic, invisible and inexpensive. We feel that the integration of some security mechanism into our network will considerably enhance the impact of our work in the outside world.

E. ARPANET AND NSW SUPPORT

During the year, our group significantly reduced the level of effort committed to maintaining the ARPANET connection to the Multics system. Although Honeywell has not officially accepted support for the ARPANET software, it has agreed that it will attempt to modify the ARPANET software when necessary as a result of changes to other parts of the system. Therefore, we are somewhat relieved of the continued effort which has been required just to maintain the ARPANET in a stable condition. The only modifications to the software that we are performing at this point are changes required to support other research activities of our group.

We continue to improve the implementation of the higher level protocols on Multics, especially the programs for sending and receiving network mail. The Information Processing Center is currently providing computer time on Multics in support of our project to produce an installable program for reading and managing mail. We are also in the process of transferring to IPC the cost of managing the system services related to receiving and sending network mail.

A significant amount of effort has been invested in making Multics a participating member of the National Software Works. At this point, Multics is a legitimate tool-bearing host in the NSW. We are in the process of transferring continued support of NSW on Multics to the Rome Air Development Center, Rome, N.Y.

Publications

1. Saltzer, Jerome. "Technical Possibilities and Problems in Protecting Data in Computer Systems." Datenschutz und Datensicherung. Edited by R. Dierstein, H. Fielder, and A. Schulz. Germany: J. P. Bachem Verlag, 1976.

2. Saltzer, Jerome. "Computer." McGraw-Hill Encyclopedia of Science and Technology. New York: McGraw-Hill 1976.

3. Svobodova, Liba. "Software Performance Monitors: Design Trade-Offs." Seventh CMG International Conference. Atlanta, Ga., November 1976.

4. Svobodova, Liba. Computer Performance Measurement and Evaluation Methods: Analysis and Applications. New York: American Elsevier, 1976.

5. Svobodova, Liba. "Computer System Measureability." Computer, Vol. 9 No. 6 (June 1976), 9-17.

6. Svobodova, Liba; Mattson, R. "The Role of Emulation in Performance Measurement and Evaluation." Proceedings of the International Symposium on Computer Performance Modelling; Measurement and Evaluation. Cambridge, Ma. March 1976.

Theses Completed

1. Benjamin, Arthur. Improving Information Storage Reliability Using a Data Network. M.I.T., Laboratory for Computer Science, LCS/TM-78. Cambridge, Ma., 1976.

2. Gifford, David. "Hardware Estimation of a Processes Primary Memory Requirements." unpublished B.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976.

3. Huber, Andrew. A Multi-process Design of a Paging System. M.I.T., Laboratory for Computer Science, LCS/TR-171. Cambridge, Ma. 1976.

4. Hunt, Douglas. A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem. M.I.T., Laboratory for Computer Science, LCS/TR-174. Cambridge, Ma., 1976.

5. Janson, Philippe. Using Type Extension to Organize Virtual Memory Mechanisms. M.I.T., Laboratory for Computer Science, LCS/TR-167. Cambridge, Ma., 1976.

6. Montgomery, Warren. A Secure and Flexible Model of Process Initiation for a Computer Utility. M.I.T., Laboratory for Computer Science, LCS/TR-163. Cambridge, Ma., 1976.

7. Reed, David. Process Multiplexing in a Layered Operating System. M.I.T., Laboratory for Computer Science, LCS/TR-164. Cambridge, Ma., 1976.

8. Shibuya, Masaoki. "Recovery for the Duplicate Database Problem." unpublished M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976.

9. Smith, Anthony. "Implementation of a Network-Wide File System on Multics." unpublished B.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976.

Theses in Progress

1. Ciccarelli, Eugene. "Multiplexed Communication for Secure Operating Systems." M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, December 1977.

2. d'Oliveira, Cecilia. "A Conjecture About Computer Decentralization." B.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, August 1977.

3. Goldberg, Harold. "A Robust Environment for Program Development." M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, February 1977.

4. Harriman, Edward. "A Microprocessor Based Implementation of a Data Stream Protocol Processor." B.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, February 1977.

5. Karger, Paul. "Non-Discretionary Access Control for Decentralized Computing Systems." M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

6. Luniewski, Allen. "A Simple and Flexible System Initialization Mechanism" M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

7. Mason, Andrew. "A Layered Virtual Memory Manager." M.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

8. Rodriguez, Humberto. "Measuring User Characteristics on the Multics System." B.S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

Talks and Presentations

1. Forsdick, Harry. "The Design of a Distributed Data Base Management System." Sperry Research Center, Sudbury, Ma., November 1976.

2. Hunt, Douglas. "Case Study of Intermodule Dependencies in a Virtual Memory Subsystem." Stanford Research Institute, Menlo Park, Ca., December 1976; C.S. Draper Laboratory, Cambridge, Ma., April 1976.

3. Janson, Philippe. "Validating the Protection Mechanism of a System." IRIA Workshop on Protection and Security in Data Networks, Le Chesnay, France, June 1976.

4. Kanodia, Rajendra. "Network Measurements." Panel member, AFIPS National Computer Conference, New York, N.Y., June 1976.

5. Kanodia, Rajendra. "Eventcounts: A New Model of Process Synchronization." Xerox Palo Alto Research Center, Palo Alto, Ca., June 1976; IBM Watson Research Center, Yorktown Heights, N.Y., June 1976.

6. Redell, David. "Proprietary Subsystems and Personal Computers." Xerox Palo Alto Research Center, Palo Alto, Ca., February 1976.

7. Redell, David. "The Multics Kernel Design Project." IBM San Jose Research Center, San Jose, Ca., March 1976.

8. Saltzer, Jerome. "Pragmatic Approaches to Obtaining Correct Operating Systems." IBM Research Laboratory, Zurich, Switzerland, September 1976; Cambridge University, Cambridge, England, September 1976; Rutgers University, New Brunswick, N.J., November 1976.

9. Saltzer, Jerome. "The Multics Kernel Design Project." Honeywell Information Systems Inc., Phoenix, Az., June 1976.

10. Saltzer, Jerome. "System Implications of Advancing Storage Technology." IBM San Jose Research Laboratory, San Jose, Ca., June 1976.

11. Schroeder, Michael. "The Multics Kernel Project." Xerox Palo Alto Research Center, Palo Alto, Ca., January 1976; Cambridge University, Cambridge, England, April 1976.

12. Svobodova, Liba. "Computer Structures." Session chairman, AFIPS National Computer Conference, New York, N.Y., June 1976.

13. Wells, Douglas. "Use of the ARPANET with Multics." Rome Air Development Center, Rome, N.Y., April 1976.

14. Wells, Douglas. "Implementation of the National Software Works on Multics." Rome Air Development Center, Rome, N.Y., April 1976.

Committee Memberships

Pogran, Kenneth. ARPA Message Service Committee

Pogran, Kenneth. ARPA Committee on Computer-Aided Human Communication

Saltzer, Jerome. ARPA IPTO Security Working Group

Wells, Douglas. ARPA IPTO NSW Working Group

DOMAIN SPECIFIC SYSTEMS RESEARCH

Academic Staff

S. A. Ward, Group Leader
M. L. Dertouzos
P. G. Jessel
J. Weizenbaum

Research Staff

C. Cesar
P. Houpt

Graduate Students

S. Y. Chiu
J. Gula
R. Halstead
A. Mok
J. Pershing
A. Reuveni
B. Schunck
T. Teixeira
C. Terman
L. Tsien
J. Wahid

Undergraduate Students

T. Hayes
D. Kahn
A. Wilding-White
E. Ziemba

Support Staff

N. MacKenzie
J. Pinella

DOMAIN SPECIFIC SYSTEMS RESEARCH

A. INTRODUCTION

During the past year the D.S.S.R. group's research activities have evolved along the dual themes of real time and distributed computing. Each of these directions is a natural consequence of the continuing effort to exploit microprocessor technology in specific applications: real time because of characteristics of typical applications, and distributed processing because of the scaling properties it affords.

B. REAL TIME BLOCK DIAGRAM SCHEMATA

This work is directed toward the implementation of a language system (compiler and run-time support) which approximates continuous-time block diagram systems on conventional general purpose digital computers. The language has been named CONSORT, standing for CONtrol Structure Optimized for Real-Time. The source language includes a description of the functional interconnection of the blocks in the diagram, and various real-time constraints that must be satisfied by the implementation. CONSORT represents a significant improvement over conventional real-time programming systems in that the user specifies an acceptable level of real-time performance without having to specify how that level of performance must be achieved i.e. a CONSORT program is a description of what to do, and not how (or more precisely, when) to do it.

Restriction of the source language to time bounded computations interconnected by fixed data paths allows scheduling strategies to be thoroughly explored at compile time, yielding in many cases a simple static control structure which guarantees the required real time performance of the object program. Such compilation involves interesting scheduling problems and, in the general case, is an NP-complete problem. Thus our approach involves compile-time heuristics and the risk of missing possible solutions to given problems.

An initial implementation (by T. Teixeira) is near completion, and generates static control structures from block diagrams with continuously varying (in time) data values. Current efforts are directed toward the automatic partitioning of schemata into sections allocated to separate processors for cases where no single processor solution can be found.

Continuing activity in this area will include further refinement of the underlying scheduling algorithms, as well as extension of the system to discrete time (and consequent production of interrupt-based control structures).

Research by P. Jessel is directed toward the development of a language which includes most traditional control structures (e.g. do.... while and conditionals) and yet

for which computation time can be bounded. The effective computation of a program module depends on the control structure of that module and on the values of its inputs. The problems of calculating estimated execution time has a number of similarities to problems in program verification. In order to estimate the execution time of a program it is necessary to trace all possible sequences and determine the time associated with each statement. The ease of this task clearly depends on the complexity of the control structures. It is a relatively easy task for linear code. However, for various control structures such as loops and conditionals the task becomes more difficult and depends on the value of the input data.

C. SEMANTICS FOR DISTRIBUTED PROCESSING

One motivation for multiple processor systems is the potential they provide for expansion without radical reorganization of problem-dependent software and techniques. Achieving this characteristic of graceful scaling requires an underlying semantics whose structure constrains as little as possible the physical locality of computations and data; we further require, of course, that this semantics be an appropriate basis for the class of computations to be performed.

In the problem domain of process control, notions of monitoring and dispatching of corrective actions upon the occurrence of certain conditions are fundamental. Abstractions of the semantics of this problem domain would suggest that parallelism is a natural state of affairs and that a dominant activity in the domain is the signalling into activity of one program module by another. In general, a program module may directly activate in parallel, multiple program modules. Symmetrically, the semantics of the problem domain also allows that the activation of a program module be dependent upon inputs from a multiplicity of program modules.

Recent works of C. Hewitt et al. [Hewitt 75] [Greif 74] [Greif 75], Kay [Xerox PARC 76] and S. Ward and Halstead [Ward 77] on message passing as a semantic basis for programming languages are especially attractive vehicles for such computations. The primitive activity in these systems is the sending of a message from one program module to another. Communication and control in these systems are not separable so that receipt of a message causes an activation of the target module with the message providing parameters for that activation. Message passing necessarily implies the use of continuations as an alternative to the implicit control return points of subexpression evaluations as found in applicative languages.

The mu-calculus has been developed by Ward and Halstead to serve as a formal semantic basis for the study of such computations in a distributed processor environment. Recent extensions give capabilities (such as the ability to produce side effects) which are desirable for modelling many practical systems. Of particular interest is the specification of tokens, a novel synchronization concept which may find application independent of the use of the mu-calculus.

Current work by Halstead has led to preliminary specifications for a distributed processor network which allows objects to move freely from one processor to another, yet enables any processor desiring to reference an object to discover an appropriate route for its request so that the request will eventually reach the object. This routing information is kept in a distributed fashion and requires only local changes if an object moves just a short distance. A distributed garbage collection algorithm allows unreferenceable objects to be detected and removed, even if the objects were at some time referenced from many different sites. A simulator for the system (running on UNIX) has been constructed.

Current work by J. Gula has led to the definition of protocols for communication between heterogeneous machines. The protocol assumes that each host machine on a network supports a network interface which transforms objects from an internal representation to a standard network representation. Interfaces support both data and procedural objects and thus one machine can specify a computation to be performed on a remote machine and supply arguments and receive results in a format consistent with local conventions.

D. AUTOMATIC CODE GENERATION

During the past year this research by Terman has concentrated on the development of a descriptive formalism to serve as the basis for the automatic creation of an optimizing code generator.

The creation of a compiler for a specific language and target machine is an arduous process. It is not uncommon to invest several years in the production of an acceptible compiler; the excellent compilers for PL/I on MULTICS and BLISS 11 on the PDP-11 evolved over a decade or more. With the rapid development of new computing hardware and the proliferation of high-level languages, such an investment is no longer practical, especially if there is little carry-over from one implementation to the next.

In an effort to automate compiler production, systems have been developed to automatically generate those portions of the compiler which translate the initial specification into an internal form suitable for code generation. These systems have enhanced portability and extensibility of the resultant compiler without a significant degradation of performance. The final phases of a compiler, those concerned with code generation, are now coming under a similar scrutiny. The ultimate goal of this research is to develop a system which can automatically construct a viable code generator. Current efforts address the issue of providing a specification of a code generator. One can envision several distinct uses for such a specification:

1. as a convenient way of replacing English descriptions of an algorithm (much the same way a BNF documents syntactically legal programs)

2. as a specification to a system which, along with a specific input string, can be interpreted in order to produce an acceptible translation (e.g. syntax directed translation based on a parse of the input string) or

3. as an input specification to a system which automatically constructs a code generator (similar to the various specifications fed to a compiler-compiler).

The extra level of interpretation (compilation in the case of a compiler-compiler) provides an added measure of flexibility not found in other code generation schemes.

The specification itself is couched in a metalanguage based on a blend of production systems, pattern matching, and attribute grammars. The basic element of the metalanguage is the form and its attributes (each attribute is an indicator-value pair). These attributes correspond to the "meaning" of their associated form; this naturally leads to two categories: inherited and synthesized attributes. Inherited attributes describe the context in which the form appears; synthesized attributes describe those properties of the form which derive from its component parts. The relationship between the attributes of one form and another is specified by "semantic rules." With sufficient care in designing the rules, it is possible to express complicated interrelationships between sets of forms as relatively simple step-by-step syntactic relationships between their components. A collection of rules can be used to describe the translation performed by a code generator and, with the inclusion of cost information, it is possible to define the optimal translation.

During the past year the syntax of the metalanguage has been finalized and a formal description of the metalanguage has been generated. The formal properties of production systems have been examined in order to determine a mechanism for translating a specification based on the above mentioned rules into an actual code generator.

Research during the coming year will be directed towards developing a sample description and compiler-compiler. Based on a survey of current optimizing code generators the metalanguage will be "specialized" to include primitive attributes that reflect common code generation techniques.

E. PROCESS CONTROL

Work in this area by P. Houpt, B. Schunck, and J. Wahid has been aimed at translating traditional analog control algorithms to digital hardware. Although this activity represents a significant improvement over the current ad hoc approach to computerized process control, restricting the domain to analog control algorithms and conventional computer structures unduly limits the solution possibilities. Accordingly one component of the group's efforts are directed at developing a more comprehensive theory of computerized control. Some of the topics under study are:

1. Timing problems: processes are most efficently utilized if they are allowed to interact asynchronously. Unfortunately, the current approach to sampled data systems implies a fixed sampling rate. For example the derivation of the equations for the LQG regulator is based on the assumption that the sampling rate is fixed in advance and remains constant while the system is under control. For a digital controller to satisfy this assumption it must be designed to guarantee that the control signal will be updated at exactly the proper instant. One solution currently being studied by Schunck is to utilize a variable sampling rate, and redefine the control equations accordingly.

2. Sampling skew: unlike analog controllers, the computation associated with the feedback loop of a digital unit significantly skews the relationship between observation and control, and in fact the observations used in one sampling period are acquired during a previous period. This violates many of the assumptions used to derive feedback gains and as a result we (Wahid) are currently attempting to incorporate this effect in the derivations.

3. The applicability of heuristic control: most control algorithms seem best suited to analog implementation. However, in implementing these algorithms on a digital processor, it seems advantageous to incorporate the inherent decision capability in the control. Our goal is to define a framework for this extended approach to control by developing a process control language with the appropriate semantic structure.

F. MICROPROCESSOR SIMULATION OF DIGITAL LOGIC

Another application of Block Diagram Schemata currently under investigation by C. Cesar is real-time simulation of conventional digital logic. The inputs to CONSORT, named in this case HOME (for Hardware on Microprocessor Emulation), are a description of the hardware and the specification of real time environmental constraints. These are independent, and are linked only by the names of input and output variables. In particular, the I/O variables are the "external variables" of the hardware description and are the basis for all real-time environment relationships. The output of the system, if emulation is possible, is the microprocessor code that simulates the hardware in real-time.

Hardware is described by a hardware description language (HDL). Our HDL is a non-procedural, single block (all variables are global), register-transfer level language. The language syntax and interpretation is tailored to the problem of real-time simulation. A "program" (i.e. a description) is composed of an unordered list of statements, where each statement is composed of an assignment prefixed by a condition. A true condition "activates" (forces execution) the assignment.

The real-time environment constraints (RTEC) description has been limited to a

set of few basic timing relations. The central idea involves defining when signal transitions (positive, leading-edge, or negative, trailing-edge) can or should occur. Transitions are located (in time) relative to other transitions, and accordingly a timing relation is specified on two transitions. If the two transitions belong to two different signals, one has an inter-signal relation. If the pair belongs to the same signal, one has an intra-signal relation. Finally, if a transition is measured relative to itself then one is defining a periodic event.

Both intra and inter-signal relations can be subdivided into two types: width and interval. Width measures (time) distances between a positive and a negative transition or between a negative and a positive transition. Interval measures distances between either two positive or two negative transitions. Periodicity is viewed as a repetitive intra-signal interval.

Absolute real-time constraints, as defined above, are too restrictive, ominously pointing to an impossible emulation. In practice, bounds on acceptable rather than absolute values are provided. These we name tolerances. Note that they are "logical"--not electrical--tolerances, and act as a bound--a maximum and a minimum--which delimits the range of possible values for a relation (width, interval, or periodicity).

The initial phase of our work is not committed to the use of a particular microprocessor architecture. The emphasis is on "proceduralization" of the aforementioned non-procedural hardware constructs. For this purpose a procedural version of the non-procedural HDL is used as the target architecture. It differs from its non-procedural cousin in two respects. First, because it is procedural, it includes extra computer control structures such as tests, jumps, and subroutine calls. Second, each operator in the HDL has a pre-defined time duration, which forms the basis of the "compilation algorithm" that derives the necessary ordering of the non-procedural constructs.

The HOME system operates on its inputs to obtain code for the hardware emulation via a three steps translation process:

1. Partial proceduralization of the non-procedural description. This involves looking for function dependencies between statements. This dependency exists when the execution of one statement can potentially cause the execution of another statement. Such dependencies indicate a desirable (faster) order for the execution of these statements. As a result of this step, a partial ordering on the statements is achieved which is independent of the timing constraints.

2. Superimposing RTEC on the partially proceduralized description. Real-time constraints are "imposed" to the partial ordering to reveal impossible emulations, to indicate further dependencies between statements, and to set up the conditions for the final translation step.

3. Final proceduralization. Using RTEC, it is now necessary to schedule statements which do not have any functional dependency and which appear, from the partial ordering of step one, to require either concurrent or parallel execution. Furthermore, RTEC helps in scheduling the acknowledgement of input changes.

G. LABORATORY

One of the first goals of our research was the development of an integrated laboratory environment which would facilitate the design of software and system tools for target microprocessors. Although this development represents an ongoing process, many of our initial goals have been achieved during the reporting period. The laboratory utilizes a PDP 11/70 running the UNIX timesharing system as the central host facility. It includes a number of conventional tools such as assemblers, simulators and downloaders for several microprocessors. In addition, during the reporting period, A. Wilding-White developed a version of BCPL for the Intel-8080 based on our partial compilation approach described in last year's report.

The facility has become a central M.I.T. resource which is used by a number of groups within the community for developing microprocessor applications. Examples include a controller for solar energy panels and a microprocessor based regulator for linear motors. In addition this facility serves as the host for all of the development work described above.

The facility also includes a hardware laboratory, coupled to the host system. We have used the laboratory primarily to demonstrate the feasibility of some of our approaches. In particular, the lab has proved to be invaluable in providing target microprocessor systems and control applications for the CONSORT project. During the period, a fourth-order inverted pendulum system was balanced.

Publications

1. Jessel, P. "Localized Microprocessor Based Networks." Proceedings of the National Telecommunications Conference. Dallas, Tx., November 28-29, 1976.

2. Jessel, P.; Chen, R.; and Patterson, R. "MININET - A Microprocessor Controlled MINI NETWORK." Proceedings of the IEEE, Special Issue on Microprocessors, Vol. 64 No. 6 (June 1976), 988-993.

3. Jessel, P., and Ward, S. "Domain Specific Systems: A New Approach to Microprocessor Based Design." EUROMICRO Symposium. Amsterdam: North-Holland, October 1976.

4. Mok, A. Task Scheduling in the Control Robotics Environment. M.I.T., Laboratory for Computer Science, MIT/LCS/TM-77. Cambridge, Ma., Sept. 1976.

Theses Completed

1. Calabi, S. "Stack Depth Distributions as Characterizations of Program Reference Strings." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976.

3. Robinson, B. "A Programmable Microprocessor System Debugger." unpublished S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, 1976.

Theses in Progress

1. Cesar, C. "Real Time Simulation Random Logic." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion May 1978.

2. Gula, J. "A Distributed Operating System for an Object Based Network." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion September 1977.

3. Halstead, R. "Multiprocessor Implementations of Message-Passing Systems." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion September 1977.

4. Pershing, J. "Design of Domain Specific Meta Compiler for Systems Using Graphical Input as a Source Language." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion September 1977.

5. Schunck, B. "Analysis of the Effect of LQG Control on Computer Structures." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion June 1978.

6. Teixeira, T. "Block Diagram Languages for Process Monitoring and Control." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion September 1977.

7. Terman, C. "A Description-Driven Universal Translator." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion September 1977.

8. Wahid, J. "Microprocessors in Control Applications." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion January 1977.

Talks

1. Jessel, P. "Second Annual Asilomar Workshop on Microprocessors." Asilomar, Ca., April 27, 1976

2. Jessel, P. "The Impact of Microprocessors on Engineering Education." Fall ASEE Middle Atlantic Section Meeting, Villanova, Pa., November 1, 1976.

3. Ward, S. "Microprocessor Software." IEEE Seminar Series, Burlington, Ma. January 1976.

4. Ward, S. "Translating the User Problem." IEEE Workshop on Microprocessors, Waltham, Ma. February 1976.

5. Ward, S. "Microprocessor Selection." IEEE Workshop on Microprocessors, Waltham, Ma. February 1976.

6. Ward, S. "Microprocessor Systems." Guest Lecture at M.I.T. Seminar Program in Artificial Intelligence, Machine Vision and Productivity, Cambridge, Ma. June 1976.

KNOWLEDGE-BASED SYSTEMS

Academic Staff

W. A. Martin, Group Leader
L. B. Hawkinson
G. R. Ruth
P. Szolovits

Research Staff

R. V. Baron
A. Boughton
G. P. Brown
G. Burke
R. Fisher
N. R. Greenfeld
D. Kapur
W. A. Kornfeld
M. K. Srivas
D. Stefanescu
A. Sunguroff

Graduate Students

R. B. Krumland
W. J. Long
W. S. Mark
M. L. Morgenstern
W. R. Swartout

Undergraduate Students

C. Kiselyak
G. Thomas

Support Staff

B. J. Demps
D. C. Foster
V. E. Lewis

KNOWLEDGE-BASED SYSTEMS

A. SUMMARY OF WORK IN PROGRESS

The research of our group may conveniently be divided into the high-level business-oriented language HIBOL, the knowledge representation system OWL, and individual knowledge representation projects.

Beginning this report with HIBOL, the summer of 1976 saw an intensive effort to get the HIBOL version of the A&T Supermarket case through the system and into compiled PL/I code. This was successful, but involved a certain amount of system handholding and did not include report formatting. In September, M. Morgenstern finished his Ph.D. thesis on the file and program configuration optimizer, and R. Baron returned to being a full-time student, doing an M.S. thesis evaluating the strengths and weaknesses of the current system. G. Ruth continued to improve various modules of the system. We are currently trying to run a version of the A&T case, including reports, through the system and to check the accuracy of the running PL/I code. We are also coding and running two other cases. From a practical point of view, the optimizer and its data requirements are the only questionable elements. We are thus developing a language for telling the system a proposed result of optimization. One could then input the HIBOL specification separately. This should make a practical language which would be fast to use and modify. The optimizer could also be used on problems of the size of A&T if desired. In his thesis, Baron is subjecting HIBOL to a careful analysis and his results should be of interest to anyone trying to design a very high level business data processing language.

On the OWL front, L. Hawkinson continued to improve his Linguistic Memory System, the module which supports the basic data structures of OWL. With the departure of A. Sunguroff, G. Brown has taken over the maintenance of the OWL I interpreter. Brown has also completed her work on the Susie Software dialogue. The decision has been made to introduce a second version of OWL, OWL II. During the past year, W. A. Martin has been working on the "world model" for OWL II, and designing an English grammar to go with it. The OWL II parser, grammar, and LMS have worked together for selected sentences and it is anticipated that the components will work well by the fall of 1977. To test these components we have sketched out a system which will be an "interactive database dictionary." This system acquires the description of the contents of databases from users and then answers questions about what data is available in the data bases with which it is familiar. Brown is implementing this system. Once this system is working well, designing a second interpreter is envisioned.

With respect to individual knowledge representation projects, W. Mark finished his Ph.D. thesis; and R. Krumland and W. Long are expected to finish shortly.

Publications

1. Mark, W. The Reformulation of Model Expertise. Ph.D. Thesis. M.I.T., Laboratory for Computer Science, LCS/TR-172. Cambridge, Ma., September 1976.

2. Martin, W. A. "Comment following article by Schank and Lehnert." Research Directions in Software Technology. Edited by Peter Wegner. Cambridge, Ma.: M.I.T. Press. To appear.

3. Ruth, G. R. "Automatic Design of Data Processing Systems." Third ACM Symposium on Principles of Programming Languages. Atlanta, Georgia, Jan. 19-21, 1976.

4. Ruth, G. R. "Automatic Programming: Automating the Software System Development Process." Research Directions in Software Technology. Edited by Peter Wegner. Cambridge, Ma.: M.I.T. Press. To appear.

Theses Completed

1. Morgenstern, M. "Automated Design and Optimization of Information Processing Systems." unpublished Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, September 1976.

Theses in Progress

1. Baron, R. B. "Structural Analysis in a Very High Level Language." M.S. Thesis, M.I.T., Dept. of Electrical Engineering and Computer Science, expected date of completion, September 1977.

2. Krumland, R. B. "Base Concepts and Mechanisms in Knowledge-Based Model Building for Managers." unpublished Ph.D. thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1977.

3. Long, W. J. "A Program Writer." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1977.

PROGRAMMING METHODOLOGY

Academic Staff

B. H. Liskov, Group Leader

Graduate Students

R. R. Atkinson
V. A. Berzins
T. Bloom
D. Kapur
M. S. Laventhal
J. E. Moss
R. N. Principato
J. C. Schaffert
A. Snyder
M. K. Srivas

Undergraduate Students

G. L. Fulton
D. P. Gorgen
E. J. McCabe
R. W. Scheifler
K. Virgile

Support Staff

M. Nieuwkerk
A. L. Rubin

PROGRAMMING METHODOLOGY

A. INTRODUCTION

The goal of the research of the Programming Methodology group is the development of tools and techniques that ease the production of quality software, software that is reliable and relatively easy to understand, modify, and maintain. Our work is based on a programming methodology in which the recognition of abstractions is the key to problem decomposition. A program is constructed in many stages. At each stage, the problem to be solved is how to implement some abstraction (the initial problem is to implement the abstract behavior required of the entire program). This is done by performing the following four steps:

1. Problem Decomposition. The programmer envisions a number of subsidiary abstractions useful in the problem domain.

2. Specification. The behavior of each abstraction is specified precisely.

3. Implementation. Once the behavior of the subsidiary abstractions is understood and specified, they can be used in a program to implement the original abstraction.

4. Verification. The programmer verifies that the implementation is correct, assuming that the subsidiary abstractions are implemented correctly.

As soon as step (2) has been performed, new problems exist concerning how to implement the abstractions defined in step (2). The programmer can choose to work on one of these problems immediately, before steps (3) and (4) have been carried out for the current stage. The process terminates when all abstractions generated during design are realized either by programs or by the programming language in use.

To make effective use of this methodology, it is necessary to understand the nature of the abstractions useful in constructing programs; this includes what is being abstracted, and what form the abstraction takes. In studying this question, we identified three kinds of useful abstractions: procedural, control and especially data abstractions. While the procedural abstraction which performs a computation on a set of input objects and produces a set of output objects, has long been recognized as useful, control and data abstractions have been neglected in discussions of programming methodology.

A control abstraction defines a method of sequencing arbitrary actions. All languages provide built-in control abstractions; examples are the **if** statement and the **while** statement. In addition, however, it is helpful to allow user definitions of a simple kind of control abstraction, which is a generalization of the repetition methods (in particular, the **for** statement) available in many programming languages. Frequently the programmer desires to perform the same action for all the objects in a collection, such

as all the characters in a string or all items in a set. The simple control abstraction permits the action to be described separately from the method of obtaining the objects in the collection.

A data abstraction is used to introduce a new type of data object that is deemed useful in the domain of the problem being solved. At the level of use, the programmer is concerned with the *behavior* of these data objects--what kinds of information can be stored in them and obtained from them. The programmer is *not* concerned with how the data objects are represented in storage, nor with the algorithms used to store and access information in them. In fact, a data abstraction is often introduced to delay such implementation decisions until a later stage of design.

The behavior of the data objects is expressed most naturally in terms of a set of operations that are meaningful for those objects. This set will include operations to create objects, to obtain information from them, and possibly to modify them. For example, push and pop are among the meaningful operations for stacks, while meaningful operations for integers include the usual arithmetic operations.

Thus, a data abstraction consists of a set of objects and a set of operations that characterize the behavior of the objects. To ensure that a data abstraction can be understood at an abstract level, we require that the set of operations *completely* determine the behavior of the data objects. This property can be achieved by making the operations the *only direct means* of creating and manipulating the objects.

The Programming Methodology group is involved in two main areas of research that support the above methodology:

1. We are developing the programming language, CLU, which provides linguistic support for programming with abstractions. Data and control abstractions are not well supported by conventional languages.

2. We are developing techniques for specifying the meaning of abstractions, and for verifying the correctness of programs written in terms of abstractions.

In the following sections we discuss some of our accomplishments of the past year. In the next section, we describe how CLU supports the use of control abstractions. (A comprehensive treatment of the abstraction mechanisms in CLU can be found in [17].) In Section C, we discuss how a language like CLU can be extended to incorporate an access control facility. Section D contains a discussion of optimization techniques for a CLU-like language. In Section E, our work on specification of data abstractions is described.

B. ITERATORS

The purpose of many loops is to perform some action on all of the objects in a collection. For such loops, it is often useful to separate the selection of the next object from the action performed on that object. CLU provides a control abstraction mechanism that permits a complete decomposition of the two activities. The **for** statement available in many programming languages provides a limited ability in this direction: it allows iteration over ranges of integers. The CLU **for** statement allows iteration over collections of any type of object. The selection of the next object in the collection is done by a user-defined *iterator*. The iterator produces the objects in the collection one at a time (the entire collection need not physically exist); the objects are then consumed by the **for** statement.

We illustrate the use of iterators by means of a simple example. Figure 1 shows an iterator called *string_chars*, which produces the characters in a string in the order in which they appear. This iterator uses string operations *size (s)*, which tells how many characters are in the string *s*, and *fetch(s, n)*, which returns the n^{th} character in the string *s* (provided the integer *n* is greater than zero and does not exceed the size of the string).

```
count_numeric = proc (s: string) returns (int);
        count: int := 0;
        for c: char in string_chars (s) do
                if char_is_numeric (c)
                        then count := count + 1;
                        end;
                end;
        return count;
        end count_numeric;

string_chars = iter (s: string) yields (char);
        index: int := 1;
        limit: int := string$size (s);
        while index <= limit do
                yield string$fetch (s, index);
                index := index + 1;
                end;
        end string_chars;
```

Figure 1. Use and Definition of a Simple Iterator.

The general form of the CLU **for** statement is

```
for declarations in iterator-invocation
    do body end;
```

An example of the use of the **for** statement occurs in the *count_numeric* **procedure (see** Figure 1), which contains a loop that counts the number of numeric characters in a string. Note that the details of how the characters are obtained from the string are entirely contained in the definition of the iterator.

Iterators work as follows: a **for** statement initially invokes an iterator, passing it some arguments. Each time a **yield** statement is executed in the iterator, the objects yielded are assigned to the variables declared in the **for** statement (following the reserved word **for**). (One or more objects may be yielded, but the number and types of objects yielded each time by an iterator must agree with the number and types of variables in a **for** statement using the iterator.) Then the loop body is executed. Next the iterator is resumed at the statement following the **yield** statement, in the same environment as when the objects were yielded. When the iterator terminates, either by an explicit **return** statement (which must not return any objects) or by completing the execution of the body, then the invoking **for** statement terminates.

For example, suppose that *string_chars* is invoked by *count_numeric* with the string "a3". The first character yielded is 'a'. At this point within *string_chars*, *index* = 1 and *limit* = 2. Next the body of the for statement is performed. Since the character 'a' is not numeric, *count* remains at 0. Next *string_chars* is resumed at the statement after the yield statement, and when resumed, *index* = 1 and *limit* = 2. Then *index* is assigned 2, and the character '3' is selected from the string and yielded. Since '3' is numeric, *count* becomes 1. Then *string_chars* is resumed, with *index* = 2 and *limit* = 2, and *index* is incremented, which causes the while loop to terminate, and the iterator to terminate. This terminates the **for** statement, with control resuming at the statement after the **for** statement, and *count* = 1.

While iterators are useful in general, they are especially valuable in conjunction with data abstractions that are collections of objects (such as sets and arrays). Iterators afford users of such abstractions access to all objects in the collection, while exposing a minimum of detail. Several iterators may be included in a data abstraction. Where the order of obtaining the objects is important, different iterators may provide different orders.

C. ACCESS CONTROL

One of the most important attributes of a programming language is the way the scope rules of the language define how data is to be shared among the individual program units (procedures, blocks, modules) out of which a program is constructed. Ordinarily, access to data is provided on an all-or-nothing basis: if a module has access to some data base, then every component of the data base is accessible, and every possible type of access (usually just reading and writing) may be performed. Experience in building large applications, or applications involving sensitive data, has indicated that sharing of data is enhanced if finer control than all-or-nothing access is provided. For example, manipulation of the information in a data base is much more controlled if not every program that reads the data base is also permitted to write it. In addition, if some of the information in a data base is sensitive, then control over which programs can read which information is also desired.

Current programming languages are deficient in providing mechanisms for controlling the sharing of information among program units. For example, passing a data base "by value" ensures that the called procedure may not modify the data base. However, this mechanism does not provide control over what parts of a data base may be read; in addition, it is so expensive for large data bases that other parameter passing mechanisms (for example, call by reference) are used instead. Proposals for avoiding the overhead of call by value while retaining the benefit that the data base cannot be modified (for example, call by reference, but permitting only read access to the formal parameter) solve the efficiency problem, but still do not provide for selective reading of the data base. In addition, such proposals do not provide for the control of selective alteration of the data base.

B. Liskov and A. Jones (Computer Science Department, Carnegie-Mellon University) have investigated a programming language extension that provides for controlled sharing of data [12]. The approach taken borrows heavily from work in operating systems, where access control mechanisms have long been one of the tools useful for realizing controlled sharing of data. In particular, our mechanism is modelled after the capability protection mechanisms provided by some operating systems [24, 26].

To incorporate an access control mechanism in a programming language, we have chosen an approach that permits programmers to express access control restrictions in terms that are meaningful to their application domains. We assume that all data are contained in *objects* for which there exists a set of *accesses*. Objects are those entities, such as data bases, libraries, stacks or files, that are of interest to programmers. Accesses are limited to those that are meaningful manipulations of the objects; accesses are the only means for altering an object or extracting information from it. In some cases, meaningful accesses are the familiar read, write, and, possibly, execute access. In other cases, the accesses themselves are user-defined, tailored to the abstract notion the user intends to capture. For example, a file system may distinguish between write access and append access. In contrast to a write access, an append

access is assumed to modify the file, but not to alter existing content. This permits a user to share a file with others, allowing them to augment the file by appending to it, but not allowing them the ability to rewrite any portion of it.

Thus, to discuss access control we require a language that permits the writing of programs in terms of data objects and the accesses that are meaningful for them. In particular, languages in which a datum is viewed as an aggregate of memory cells, are not suitable, because of the difficulty of expressing access control on anything but a cell basis. One class of languages, including the languages SIMULA 67 [3, 4], CLU [17], and Alphard [28], provides a natural environment in which to embed an access control facility. In these languages, a data type is viewed as a set of objects and a set of operations. The operations of a data type correspond very closely (though not identically, as we shall show) to our notion of access, and access control corresponds to the ability to control the use of the operations.

To accommodate access control, we will add one more component to a type: in addition to objects and operations, a type also specifies a set of rights. A *right* is a name that represents a meaningful manipulation of objects of the type; often a right corresponds to the use of one of the type's operations. The basic idea behind rights is: to legally apply one of the type's operations, a user must hold appropriate rights to the objects passed to that operation as parameters.

An example is given in Figure 2 for the type, *AssociativeMemory*. Operations for this type include an operation to create an empty *AssociativeMemory* object of a particular size (*makemem*), an operation to add a name-value pair to an *AssociativeMemory* (*insert*), an operation to change the value associated with a given name (*change*), an operation to fetch the value associated with a given name (*getval*), and an operation to remove a name-value pair (*delete*). In order for *insert*, *change*, *getval*, or *delete* to be invoked, the invoker must present a right to apply the operation to the *AssociativeMemory* object passed in as a parameter; in this particular example, the name of the required right is the same as the name of the operation. The *makemem* operation returns all these rights for the *AssociativeMemory* object it creates. The *AssociativeMemory* operations also use objects of type *integer*; for simplicity we have chosen to omit information about required rights for all *integer* objects. In general, we can expect some rights to correspond to the use of a single operation, some to a group of operations and some to a single parameter of an operation taking more than one object of the type.

Embedding an access control facility in a programming language permits expression of access restrictions as an integral part of a program. In addition, the question of whether a program obeys access control restrictions, and is thus *access-correct*, can be answered at compile time. This can lead to benefits similar to those derived from compile-time type checking: confidence that the program is access-correct, and enhanced efficiency over the dynamic mechanisms currently provided by operating systems.

```
type: AssociativeMemory
rights: "insert", "change", "getval", "delete"
operations:

  makemem
    input:  integer;                (desired AssociativeMemory size)
    output: AssociativeMemory;  "insert","change","getval", "delete" rights are given

  insert
    input:  AssociativeMemory;  "insert" right required
            integer;            the name)
            integer;            the value)
    effect: (insert modifies its AssociativeMemory parameter)

  change
    input:  AssociativeMemory;  "change" right required
            integer;            (the name)
            integer;            (the new value)
    effect: (change modifies its AssociativeMemory parameter)

  getval
    input:  AssociativeMemory;  "getval" right required
            integer;            (the name)
    output: integer;            (the value)

  delete
    input:  AssociativeMemory;  "delete" right required
            integer;           (the name)
    effect: (delete modifies its AssociativeMemory parameter)
```

Figure 2. The AssociativeMemory Type.

1. Basic Model

Our approach to access control is based on a semantic model in which *objects* are shared among *variables*. Each object has a *type*, which determines the legal accesses to the object. Our notation for access control involves a declaration for each variable of the type of object that variable may refer to, and the rights that are available for that object when it is used via the variable. These two pieces of information are captured in the notion of a *qualified type*. A qualified type is written

T{r1,...,rn}

where T is the name of some type, and $\{r1,...,rn\}$ is a non-empty subset of the rights of T. We refer to the two parts of a qualified type as the base type and the rights; if Q is a qualified type, then $base(Q)$ is the base type and $rights(Q)$ is the rights. For example, the following are some of the qualified types derived from AssociativeMemory

AssociativeMemory {getval}
AssociativeMemory {insert, change}
AssociativeMemory {insert, change, getval, delete}

The final example specifies all the AssociativeMemory rights; a special notation

T{all}

may be used instead of listing all the rights.

Qualified types are used in variable declarations and in formal parameter specifications in procedure headings. An example of a variable declaration is:

v: AssociativeMemory {insert, change}

The meaning of this declaration is: v is a variable that can be used to refer to *AssociativeMemory* objects, but only the "insert" and "change" rights may be exercised in conjunction with v.

We view a variable as a pair

{object id, qualified type}

The object id is a unique name that is interpreted by the underlying addressing mechanism to select an object. When a variable is created, its qualified type is defined once and for all and can never be altered. However, the object named by a variable (via the object id) can change by application of the *binding* operation. Binding causes a variable to refer to an object by storing that object's id in the variable. Note that it is possible for sharing of objects to take place, because two variables may contain the same object id. In this case, the qualified type in the two variables may differ, but the binding rule (discussed in the next section) ensures that the base type is necessarily the same.

A variable contains a capability in the operating system sense [5, 14]. The capability provides the basis for restricting the kinds of manipulation that can be performed on the object specified by the object id. Intuitively, the restrictions on how an object can be used are expressed along the path to the object (the path through the object id in the variable). Thus, using one path rather than another to name an object changes the way the object can be manipulated. For example, suppose

a: AssociativeMemory{getval, insert}
b: AssociativeMemory{getval}

both name the same object. Using *b* it is impossible to modify this object, since only the *getval* operation can be used; using *a*, the object may be modified by application of the *insert* operation.

Our notions of variable, object and binding are different from the related notions of value and assignments that underlie block-structured languages. This difference is illustrated in Figure 3. Figure 3a shows the traditional view of variables and values, in which the value resides in the variable and a new value can be copied into a variable by means of assignment. Figure 3b illustrates our semantics: a variable is bound to an object, and a value is contained in an object. This value may be accessed or modified only by means of one of the operations of the object's type. Our rule of binding differs from assignment in that it causes *sharing* of the object involved, rather than the copying of the value in the object. Furthermore, this sharing is significant since, for some types of objects, operations exist to change the value inside of the object. For example, the *AssociativeMemory* operations *insert*, *change* and *delete* modify the value inside of an *AssociativeMemory* object.

Our notion of binding corresponds to assignments involving variables holding (typed) references to objects. Some programming languages are based on a semantic model like ours. The most widely known of these languages is LISP [18]; LISP lists are objects (with operations car, cdr, and cons) and LISP setq is similar to our binding. Our model is also used in SIMULA 67 and CLU.

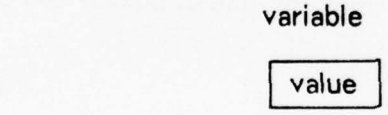

Figure 3a. Traditional view of variables and values.

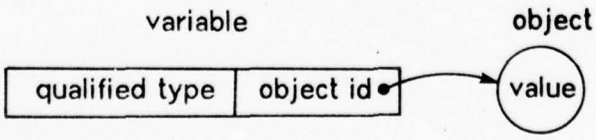

Figure 3b. Model used in this paper.

Figure 3. Comparison of Semantic Models.

We believe that our semantics models very well what is going on in systems where controlled sharing is of interest. Note that sharing of objects is a fundamental fact in these systems; the sharing of actual objects (rather than just copies of the values of objects) leads both to interesting behavior (e.g. many programs working with the same data base), and the need to exercise some control over exactly how the object should be shared. Protection schemes exist to provide this control.

2. Binding Rule

A single rule, governing the legality of binding of objects to variables, is sufficient to provide the required access control and is the basis for determining whether a program is access-correct (obeys the access control restrictions). Binding is the operation that causes a variable to refer to an object (by changing the object id). The effect of binding is creation of a new access path for the object. Therefore, to ensure that a program is access-correct, we must guarantee that no new access rights to the object are obtained from this new access path. For example, suppose that x and y are variables, and that x is to be bound to the object currently bound to y. This new binding should be allowed only if the qualified types of x and y both arise from the same base type, and if the rights obtainable by referring to the object via variable x do not exceed the rights obtainable by referring to the object via y.

We can formalize this rule as follows. First, we define what it means for one qualified type to be greater than or equal to another. *If $Q1$ and $Q2$ are qualified types,* then $Q1$ is *greater than or equal to* $Q2$, written

$$Q1 \geq Q2$$

if $base(Q1) = base(Q2)$ and $rights(Q1)$ rights$(Q2)$. Now the rule of binding can be defined:

$$v \leftarrow e$$

where v is a variable and e is an expression and

T_v = qualified type of variable v
T_e = qualified type of expression e

is legal provided that

$$T_e \geq T_v$$

Thus a binding is legal only if the new access path provides at most a subset of the rights obtainable via the original access path. Note that this rule ensures that a variable will always refer to an object whose type is the base type of the qualified type of the variable.

An expression is either a variable, in which case its qualified type is the same

as the qualified type of the variable, or it is a procedure invocation. In the former case, we have now defined the rule of binding (since T_e is the qualified type of this variable). For example, suppose

a. AssociativeMemory{getval, insert}
b. AssociativeMemory{getval}

Then $b \leftarrow a$ is legal, but $a \leftarrow b$ is not. This is illustrated in Figure 4. In Figure 4a, an initial configuration is shown in which a refers to an AssociativeMemory object α, and b refers to an AssociativeMemory object β. Figure 4b shows the result of $b \leftarrow a$. Both b and a now refer to α. A new access path (from b to α) has been created as a result of this binding, but no new rights to α are obtained by it; in fact, the new access path via b has fewer rights to α than the old access path. Figure 4c illustrates what would be the result of $a \leftarrow b$. If this binding were allowed, the new access path from a to β would allow more rights than the old one, and therefore the binding must not be permitted.

In order to understand binding when the right-hand side is a procedure invocation, we must examine the semantics of parameter passing. Our notion of parameter passing is defined in terms of binding. A procedure definition has the form

```
procedure <procname> (<formals specification>)
        returns <result specification>
        <body>
        end <procname>
```

where <formals specification> specifies the name and qualified type for each formal parameter, and <result specification> specifies the qualified type returned by the procedure. Each formal parameter is considered to be a local variable of the procedure; this variable is created at invocation, and the actual parameter is bound to it. The procedure invocation is legal only if the bindings of actual to formal parameters are legal. The qualified type of the invocation expression is then the type specified in the <result specification>.

For example, suppose a procedure *P* has type requirements

```
procedure P (x: T1{f1,f2}) returns T2{g1}
```

and declarations

```
a: T1{f1,f2,f4}
b: T2{g1}
```

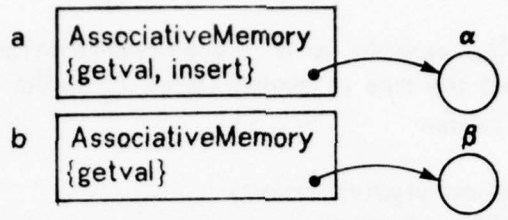

Figure 4a. The initial state.

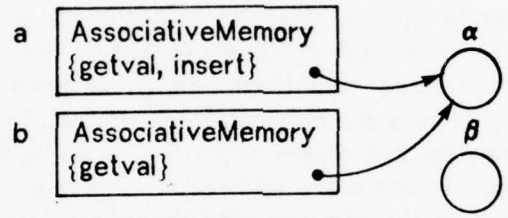

Figure 4b. Result of b ← a.

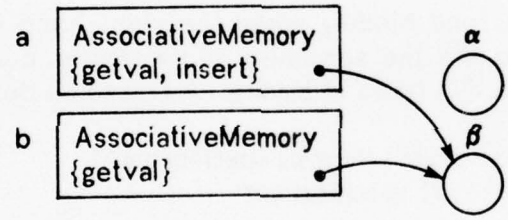

Figure 4c. Result of a ← b (disallowed).

Figure 4. Binding.

occur in the invoker of *P*. Then the statement $b \leftarrow P(a)$ is legal because the invocation $P(a)$ is legal ($x \leftarrow a$ is legal), and the object returned by *P* has qualified type $T2\{g1\}$ and therefore may be legally bound to *b*. However, $b \leftarrow P(c)$, where $c: T1\{f1,f3\}$, is not legal because the invocation $P(c)$ is not legal ($x \leftarrow c$ is not legal).

The question of whether a procedure definition is access-correct can be answered independently of any invocation of that procedure. A procedure is access-correct provided that all bindings within it are legal, and that for every return statement:

return <expr>

the qualified type of <expr> is greater than or equal to the qualified type in the procedure <result specification>.

Procedure invocation is the mechanism whereby objects are created in the first place. There exist a number of primitive data types (for example *integer, boolean, array*). The creation operations of these types provide objects of the type whenever they are invoked, and these objects are returned with full rights. For the non-primitive, user-defined types the situation is analogous. This has already been illustrated in the *AssociativeMemory* example shown in Figure 1; whenever the *makemem* operation for *AssociativeMemory* is invoked, it returns a new *AssociativeMemory* object with full rights. Thus the creator of an object obtains all rights to it. As the object is passed from one access-correct procedure to another, certain rights may be removed, but rights are never gained. This is true because binding is the only method provided for sharing objects between procedures.

3. Discussion

The access control mechanism described above is sufficient to control the sharing of many of the kinds of objects of interest in programming. For example, suppose we define a type *employee-record*, with operations (and rights) to *read-job-category*, *write-job-category*, *read-salary*, and *write-salary*, among others. Using the rules defined so far, we can define a procedure

procedure P (x: employee-record{read-job-category, write-salary})

which computes a new salary based on the employee's job category, but is unable to change the job category, or to read the old salary.

The above discussion is intended to introduce the reader to the access control facility. A complete description of this facility, which includes the following additional topics, is given in [12]:

a. The use of *amplification* [10] in the program module defining a new type.

b. An extension of the binding rule to control sharing of objects passed indirectly--through the medium of another object.

c. A comparison of the access control facility with the dynamic mechanism present in the Hydra system [11, 16].

D. OPTIMIZATION

One objection raised to the adoption of structured programming methods is that they produce inefficient programs. While we believe that the major cost of software is its construction and verification, the cost of executing programs cannot be ignored. Both costs can be reduced by the use of program optimization techniques. The rationale for program optimization is nicely stated by W. Wulf, et al. [27, p. 131],

The reason that compiler optimization is important is that programmer efficiency and execution efficiency need not be a choice we must make. Optimization is a technological device to let us have our cake and eat it, too--*both* to have convenient and well-structured programming and efficient programs.

R. Atkinson has investigated an approach to optimization that is especially applicable to languages like CLU [1]. First a program is transformed by a technique known as inline substitution, which substitutes the bodies of procedures for certain invocations of those procedures. This transformation tends to increase the size of the transformed program, but tends to decrease the execution time by eliminating procedure call overhead, and by enabling more global optimizations. Then the data and control flow of the transformed program is obtained using symbolic interpretation. Finally, standard optimization techniques, such as constant propagation, are performed, making use of the data and control flow information and, in addition, information about properties of procedures and about the interaction among the operations of a data abstraction.

1. Inline Substitution

Inline substitution reduces execution time by eliminating the overhead involved in using the procedure call mechanism. The size change resulting from a substitution is simply the difference between the size of the expanded invocation and the size of that part of the call mechanism originally present in the code. Coupled with these "direct" effects on space and time are corresponding "indirect" effects. Placing a procedure body in a specific context can present new opportunities for optimization using other techniques. These optimizations will generally reduce execution time even further, but their effect on program size will depend on the technique.

When procedure bodies are small, as they are in CLU programs, many optimization techniques are ineffective, simply because they require the presence of a substantial context. Thus, performing inline substitution before using other techniques may be the key to successful optimization of structured programs.

R. Scheifler has studied inline substitution as an independent optimization technique [23]. This study involved the analysis of the following problem: given a program and constraints on the final program size, find a sequence of substitutions that minimizes the expected execution time, considering only "direct" effects.

A key phrase in this problem statement is "expected execution time." Some method is needed to determine the number of times an invocation is expected to execute. We believe a good method is to run the program using data selected by the programmer, and to count the number of times each invocation executes. These statistics can then be used as the initial expected numbers. They are "initial" numbers for two reasons:

a. Inline substitution can create new invocations, each of which must be assigned an expected number.

b. When the body of a procedure P is substituted for an invocation, P is no longer called as often, implying that new expected numbers must be assigned to invocations contained in P.

To completely determine how expected numbers change, the control flow history must be retained in the statistics, necessitating many counters for each invocation. However, a single counter will suffice if a simplifying assumption is made about control flow: for any procedure body and any invocation contained therein, the expected number of executions of the invocation per execution of the body is constant. From this assumption a set of equations has been developed for calculating new expected numbers. The equations work when substituting for recursive as well as non-recursive invocations.

Using these equations, an algorithm to perform inline substitution can be formulated. However, as a practical matter, the problem of finding a set of substitutions that minimizes execution time is intractable. R. Scheifler has shown this problem to be NP-hard, meaning there is no known algorithm that will always solve the problem in polynomial time, and the existence of such an algorithm would imply polynomial-time algorithms for many classic hard problems [23].

An approximate solution to the problem has been developed, and is implemented for the current CLU system. The algorithm is built on a very simple heuristic: substitute for invocations that execute often but call small procedures. More precisely, at each step choose the invocation that will yield the greatest time savings per unit space increase. Continue until the maximum program size is reached. Lastly, while there is an invocation that is the sole remaining invocation of a non-recursive procedure, substitute for the invocation. This allows the procedure itself to be discarded, and so does not increase the program size.

Preliminary results using this algorithm indicate that, in programs with a low degree of recursion, over 90 percent of all procedure calls can be eliminated with little increase (-1 to 25 percent) in the size of compiled code, and with moderate savings (10 to 30 percent) in execution time.

2. Program Analysis

Following inline substitution, two kinds of program analysis are carried out. First, the program is analyzed to obtain information about its control flow and data flow. Then the flow information is analyzed to identify potential optimizations.

R. Atkinson has investigated a non-standard method for obtaining control and data flow information [1]. He has adapted the technique of *symbolic interpretation* [13], in which a program is executed using symbolic objects rather than actual objects.

Symbolic interpretation can be used to obtain both data and control flow information.

As an example of obtaining data flow information, suppose we have the procedure:

```
square = proc (x: int) returns (int);
        return x * x;
        end square;
```

The symbolic interpretation would start by associating a symbolic object (#1) with the variable x. Then the integer multiply operation would be interpreted to obtain another symbolic object (#2 = int$mul(#1, #1)). The object returned by the procedure is #2. The symbolic interpretation removes our dependence on variables, so that we are only concerned with the symbolic objects.

After performing symbolic interpretation on the program, the optimizer searches for transformations that will make the program less costly to execute. One such transformation is the replacement of redundant expressions by variables that hold previously calculated objects. The method used is to search the set of symbolic objects created by the symbolic interpretation for equivalent symbolic objects; then the control flow information provided by the symbolic interpretation is used to discover whether the calculation of one of the objects precedes the other. For example,

```
u := a[i]
  ...
v := a[i]
```

where a is an $array[t]$, for some type t, and i is an integer, can be transformed into

```
u := a[i]
  ...
v := u
```

provided that in the intervening code there are no assignments to variables u, a and i, and there are no side-effects that affect the equivalence of the objects in variables u and v. If u and v are found to contain equivalent symbolic objects, this guarantees that none of u, a and i have been assigned to in the intervening code. To determine whether a side effect has occurred, the optimizer requires information about the properties of the data and procedural abstractions used in the program being optimized. For example, the only side effect that could invalidate the substitution shown above is to update the nth element of the array object referred to by a. Thus, the information that use of the array update operations can affect the later use of the array fetch operation $a[i]$ constitutes a property of arrays that is of interest to the optimizer. (In CLU, $a[i]$ is not considered to be a variable, but rather syntactic sugar for an invocation of an array operation. If $a[i]$ appears on the right hand side of the assignment symbol, it stands for a call on the array fetch operation; if it appears on

the left hand side, it stands for a call on the array store operation. The reader is referred to [17] for an explanation of CLU semantics.)

3. Determining Properties of Abstractions

Some properties of data and procedural abstractions that we have found useful for optimization follow:

a. *mutability*: an object is mutable if the information in it can change over time, and immutable if all of its information is constant over time. A data abstraction is immutable if all of its objects are; otherwise the data abstraction is mutable. Integers and strings are immutable in CLU, while arrays and records are mutable.

b. *isolated representation*: a data abstraction has an isolated representation if the objects of that data abstraction can only be modified through operations of the abstraction.

c. *obscuring*: procedure P obscures procedure Q if the execution of P modifies an object and Q uses the modified component.

d. *side-effect free*: a procedure P is side-effect free if executing P does not modify any objects existing prior to its execution. All procedures that implement mathematical functions are side-effect free, as well as many procedures that examine mutable objects.

The optimizer design we have proposed can use properties about abstractions. We assume these properties are computed prior to optimization and are stored in a data base. In general, however, it is costly (and sometimes impossible) to determine such properties. Therefore, R. Atkinson [1] has developed techniques that provide conservative approximations to the desired properties. Where the properties cannot be determined, worst-case assumptions are made (for example, if a data type cannot be shown to have immutable objects, the optimizer must assume that the objects are mutable).

In making these approximations, we depend on the notion of reachability for CLU objects. The only objects reachable are those in some basis set (such as the parameters passed to a procedure), or those objects that are reachable from other reachable objects. We call the set of objects that are reachable from some object X the *reachablity closure* of X.

Unfortunately, the reachability closures for mutable objects are dynamic, and cannot generally be determined prior to execution. We can approximate reachability closures, however, by noting that CLU data types partition the set of all CLU objects in such a way that objects in different partitions can never be reached from one another. Furthermore, a static structure *does* exist for CLU data types (once implementations have been selected for these types). We therefore define a *type closure* of an abstract

type T to be the set containing T and all types in the type closure of the representation type of T (the type chosen to represent objects of type T, and referred to within a cluster implementing T as the rep--see [17] for more information). The type closure of a basic type B (such as *integer, boolean, string, array*[...], and *record*[...]) is the union of the type closures of the type parameters to B and the set containing only B. As an example, the type closure of *array*[*integer*] is {*array*[*integer*], *integer*}. As a second example, suppose that *array*[*integer*] is the representation type of the abstract type *stack*[*integer*]. Then the type closure of *stack*[*integer*] is {*stack*[*integer*], *array*[*integer*], *integer*}.

Given an object X of type T, then the type of every object in the reachability closure of X is in the type closure of T. For example, from any object of type *array*[*integer*] only objects of type *integer* or *array*[*integer*] can be reached, while from a *stack*[*integer*] object, only objects of type *stack*[*integer*], *integer* or *array*[*integer*] can be reached.

The use of type closures may be illustrated by returning to our earlier example. Suppose the actual code segment was

```
u := a[i]
p(x, y)
v := a[i]
```

where x: S and y: R. If the union of the type closures of S and R does not include *array*[t], then we can be certain that a is not modified in p, since a cannot be reached from either x or y.

Other closures can be constructed in much the same way as type closures. Two closures defined on procedures are the *mutability closure* and the *access closure*. The mutability closure of procedure P is the set of all types with mutable objects that can be changed during an execution of P. The access closure of procedure Q is the set of all types examined during an execution of Q. As with the type closure, these closures are ultimately derived from known properties of the basic CLU types. The mutability and access closures can be used to approximate the obscuring property for P and Q. We assume that P obscures Q if the intersection of the mutability closure of P with the access closure of Q is not the empty set.

Use of the obscuring property may permit optimizations that would be forbidden if only type closures were considered. In the example above, if the mutability closure of procedure p does not contain *array*[t], then p does not obscure the first array fetch operation and therefore the second array fetch operation can be eliminated. This may occur even if *array*[t] were contained in the union of the type closures of S and R.

Not all properties useful to the optimizer can be approximated with closures. For example, using the above methods, we may be able to determine that the data abstraction, *stack*[t], with operations *push, pop, top, size* and *equal*, has the following

properties:

stack[t] objects are mutable
stack[t] has an isolated representation
top, size, and equal are side-effect free
push obscures top, size
pop obscures top, size

One additional property of interest would express the fact that push (or pop) only obscures top (or size) if the same stack object is given to both push and top. A further property expresses information about equivalence of symbolic objects. For example, after *push(s, v)*, we know that $v = top(s)$. Information of this sort could be used during program transformation to avoid the *top(s)* computation, and use a previously computed object.

Although closures cannot be used to approximate every property of interest, a considerable amount of information can be obtained from their use. Such information is needed for optimizing languages, like CLU, that provide data abstractions. The information would also be useful for optimizing programs with pointers.

E. SPECIFICATIONS FOR DATA ABSTRACTIONS

There are three methods for specifying data abstractions [15, 16]: axiomatic, state machine, and abstract model.

The most promising form of axiomatic specification is the algebraic technique, developed by Zilles at M.I.T. [29], using some results in algebra [2]. The technique was investigated further by Guttag at the University of Toronto [6], who worked out a criterion for recognizing a "sufficiently complete" axiomatization of a data type. Further work on verification of data types using this technique is in progress at ISI [7, 8].

The state machine approach was first proposed by Parnas [20]. The approach as originally proposed was informal. Work on formalization of this technique is underway [21, 22].

The abstract model approach has been used informally in [9]. During the past year, we have been studying the formalization of this technique. Some work in this area has also been done by Wulf et al. [28].

In [15], we developed some criteria for judging the desirability of a specification technique for data abstractions. Among the criteria were the ease of construction and understandability of the specifications. We believe that the abstract model specification technique is best with respect to these criteria; this is the motivation for our work on this technique.

properties:

stack[t] objects are mutable
stack[t] has an isolated representation
top, size, and equal are side-effect free
push obscures top, size
pop obscures top, size

One additional property of interest would express the fact that push (or pop) only obscures top (or size) if the same stack object is given to both push and top. A further property expresses information about equivalence of symbolic objects. For example, after $push(s, v)$, we know that $v = top(s)$. Information of this sort could be used during program transformation to avoid the $top(s)$ computation, and use a previously computed object.

Although closures cannot be used to approximate every property of interest, a considerable amount of information can be obtained from their use. Such information is needed for optimizing languages, like CLU, that provide data abstractions. The information would also be useful for optimizing programs with pointers.

E. SPECIFICATIONS FOR DATA ABSTRACTIONS

There are three methods for specifying data abstractions [15, 16]: axiomatic, state machine, and abstract model.

The most promising form of axiomatic specification is the algebraic technique, developed by Zilles at M.I.T. [29], using some results in algebra [2]. The technique was investigated further by Guttag at the University of Toronto [6], who worked out a criterion for recognizing a "sufficiently complete" axiomatization of a data type. Further work on verification of data types using this technique is in progress at ISI [7, 8].

The state machine approach was first proposed by Parnas [20]. The approach as originally proposed was informal. Work on formalization of this technique is underway [21, 22].

The abstract model approach has been used informally in [9]. During the past year, we have been studying the formalization of this technique. Some work in this area has also been done by Wulf et al. [28].

In [15], we developed some criteria for judging the desirability of a specification technique for data abstractions. Among the criteria were the ease of construction and understandability of the specifications. We believe that the abstract model specification technique is best with respect to these criteria; this is the motivation for our work on this technique.

In the remainder of this section, we discuss the work of V. Berzins on the abstract model technique. He has worked out the theoretical justification for this technique (which is also algebraic in nature). He has investigated the structure of the specifications and has arrived at a form that, we believe, makes it easier to build specifications. He has also developed criteria for establishing consistency and completeness of abstract model specifications (analogous to those developed by Guttag [6] for algebraic specifications). These criteria are helpful in evaluating the specification of an abstraction, since a specification that is not well formed cannot define any behavior, let alone the intended behavior.

1. Abstract Model Specifications

A sample specification using the abstract model technique is shown in Figure 5. A sequential file data type is defined, which can be written in a restricted way: records can only be appended to a file, but not deleted or updated. The files are sequential because they can only be scanned by starting at the beginning and spacing forward.

An abstract model specification has three major parts, describing the interface, the abstract representation, and the operations of the data type.

The interface of a data type consists of the names, domains, and ranges of its operations. This information is singled out because the operations provide the sole access to the abstract objects of the type. Thus a program, a proof, or even the rest of the specification can be checked for type correctness using only the information contained in the interface specifications of the data types that are used. (This is precisely the information that must be provided whenever abstractions are added to the CLU system, and the CLU compiler checks all uses and implementations of an abstraction for consistency with this information.)

The *abstract representation* is introduced into the specification solely to provide a framework in which to define the behavior of the operations of the type, and does *not* constrain the class of representations that may be used in the implementation. The types used in the abstract representation are chosen for simplicity rather than for efficiency. The primary use of specifications is for communication, and (perhaps) in proofs of program properties; how well they run as programs is of secondary interest. Therefore simplicity and clarity are important, while hypothetical time and space requirements are not.

The abstract representation has three subcomponents in its specification: the representation type, the abstract invariant, and the abstract equivalence relation. The representation type must be composed from previously defined types. We favor using finite sets, sequences, and tuples to put together known types into new ones. (Although we have not included them in this report, formal, axiomatic definitions of these families of types have been developed.)

```
Type FILE[RECORD] is

Interface:
      create() --> FILE,
      append(FILE, RECORD) --> FILE ∪ {error(append-in-middle)},
      reset(FILE) --> FILE ∪ {error(file-empty)},
      skip(FILE, int) --> FILE ∪ {error(skip-past-eof), error(reverse-skip)},
      read(FILE) --> RECORD ∪ {error(file-empty)},
      eof(FILE) --> bool,

Representation:   tuple[ptr: int, s: sequence[RECORD]],

      Invariant:    For all f: FILE;
            0 ≤ f.ptr ≤ length(f.s) & (length(f.s) > 0 ==> f.ptr > 0),

      Equivalence:  For all (f1, f2): FILE;
            f1 = f2 <==> (f1.ptr = f2.ptr & f1.s = f2.s),

Operations:    For all (f, f1, f2): FILE, r: RECORD, n: int;
      create() = tuple[ptr: 0, s: emptyseq()],
      append(f, r) = if f.ptr = length(f.s) then tuple[ptr: f.ptr + 1, s: addlast(r, f.s)]
                     else error(append-in-middle),
      reset(f) = if length(f.s) > 0 then tuple[ptr: 1, s: f.s]
                     else error(file-empty),
      skip(f, n) = if n < 0 then error(reverse-skip)
                     else if f.ptr + n > length(f.s) then error(skip-past-eof)
                     else tuple[ptr: f.ptr + n, s: f.s],
      read(f) = if f.ptr = 0 then error(file-empty)
                     else nth(f.ptr, f.s),
      eof(f) = f.ptr = length(f.s),
end type.
```

Figure 5. Sample Abstract Model Specification.

Every meaningful abstract object should have a unique abstract representation, and conversely. The *invariant* describes a restriction on the representation type which excludes those elements that do not represent any meaningful abstract object. (It is similar in this respect to the invariant of the concrete representation [9] used in proving the correctness of an implementation of a data abstraction.) The *equivalence* is a relation stating which pairs of the representation type represent the same abstract object. If there are multiple meaningful representations for each abstract object, we can take the entire set (equivalence class) of elements representing an abstract object

to be its unique abstract representation. The abstract equivalence is important because it specifies precisely which properties of the representation are being used to *model* the abstract type.

In the example, the state of a file is represented by a sequence of records, and a pointer into that sequence to indicate which record is currently being scanned. Note that the pointer is a natural number, which by definition cannot be negative, although it can be zero. The invariant says that the pointer can never get past the end of the sequence, and that provided the file is not empty, the pointer will always point at some record of the sequence (the first record has index 1). The equivalence tells us that each object of the representation type satisfying the invariant represents a unique file object.

The operations are defined as functions on the representation type, in as simple and clear a way as possible (efficiency does not matter). Any formal method for defining functions is acceptable. We will use both McCarthy's recursive conditional expressions [19], and input/output constraints expressed in the predicate calculus, as we find most convenient.

In the example, all of the operations except for *eof* are defined using conditional expressions, none of which need be recursive because of the simplicity of the data abstraction. *Eof* is defined as a predicate on the representation type, which happens not to require conditionals or quantifiers.

2. Consistency and Completeness of Abstract Model Specifications

A specification describes the behavior of some abstraction, and it is important that it describe that behavior correctly. While it is clearly not possible to *prove* that the specification is correct, it is possible, by analyzing properties of the specification, to identify problems, or alternatively to gain confidence in the correctness of the specification. Guttag [6] has done some work along these lines for algebraic specifications. We discuss below some criteria for abstract model specifications that we have developed for this purpose.

A well formed abstract model specification must satisfy the following requirements:

a. Type Correctness. The definitions of the operations must be consistent with the interface specifications, and all expressions of previously defined types must be consistent with the interface specifications of those types.

b. Representation consistency.

   1. The invariant must be a well formed unary predicate on the representation type.

2. The equivalence must be a well formed binary predicate on the representation type, and it must define an equivalence relation (it must be reflexive, symmetric, and transitive).

c. Totality. Every operation mentioned in the interface specification must be uniquely defined for all elements of the representation type satisfying the invariant relation.

d. Closure. Every element in the intersection of the range of an operation with the representation type must satisfy the invariant relation.

e. Congruence. Every operation must be consistent with the representation equivalence, which means that equivalent inputs must result in equivalent outputs.

Some of these requirements are easier to check than others. The bulk of the type correctness check can be performed by a fairly simple algorithm, such as the one used by the CLU compiler. (Showing that no error values are produced, except for those described in the interface specifications, may require some program analysis.) At the other extreme, deciding whether a recursive function is total is undecidable in the general case, although there are well known techniques for proving termination, which apply to most programs that are designed to terminate [25]. A moderately powerful theorem proving facility is needed to demonstrate that all the requirements are met, comparable to the facility required for verifying that programs meet their specifications.

REFERENCES

1. Atkinson, Russell R. "Optimization Techniques for a Structured Programming Language." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1976.

2. Birkhoff, Garrett, and Lipson, John D. "Heterogeneous Algebras." Journal of Combinatorial Theory, Vol. 8 No. 1 (January 1970), 115-133.

3. Dahl, Ole-Johan; Myhrhaug, B.; and Nygaard, Kristen. The SIMULA 67 Common Base Language. Norwegian Computing Center, Publication S-22, Oslo, Norway 1970.

4. Dahl, Ole-Johan, and Hoare, C. A. R. "Hierarchical Program Structures." Structured Programming. Edited by O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. New York: Academic Press, 1972.

5. Dennis, Jack B., and van Horn, Earl C. "Programming Semantics for Multiprogrammed Computations." Communications of the ACM, Vol. 9 No. 3 (March 1966), 143-155.

6. Guttag, John V. The Specification and Application to Programming of Abstract Data Types. Ph.D. Thesis, University of Toronto, Computer Systems Research Group, CSRG-59. Toronto, Canada, 1975.

7. Guttag, John V. "Abstract Data Types and the Development of Data Structures." Communications of the ACM, Vol. 20 No. 6 (June 1977), 396-404.

8. Guttag, John V.; Horowitz, Ellis; and Musser, David R. Abstract Data Types and Software Validation. University of Southern California, Information Sciences Institute, Report ISI/RR-76-48, Los Angeles, Ca., 1976.

9. Hoare, C. A. R. "Proof of Correctness of Data Representations." Acta Informatica, Vol. 1 No. 4 (1972), 271-281.

10. Jones, Anita K. Protection in Programming Systems. Ph.D. Thesis, Carnegie-Mellon University, Department of Computer Science, Technical Report. 1973.

11. Jones, Anita K., and Wulf, William A. "Towards the Design of Secure Systems." Software Practice and Experience, Vol. 5 No. 4 (October-December 1975), 321-336.

12. Jones, Anita K., and Liskov, Barbara H. An Access Control Facility for Programming Languages. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 137. Cambridge, Ma., April 1976.

13. King, James C. Symbolic Execution and Program Testing. IBM Thomas J. Watson Research Center, RC 5082. Yorktown Heights, N. Y., October 1973.

14. Lampson, Butler W. "Protection." Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems. Princeton University, 1971, 437-443.

15. Liskov, Barbara H., and Zilles, Stephen N. "Specification Techniques for Data Abstractions." IEEE Transactions on Software Engineering, Vol. SE-I No. 1 (March 1975), 7-19.

16. Liskov, Barbara H., and Berzins, Valdis. An Appraisal of Program Specifications. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 141-1. Cambridge, Ma., July 1976.

17. Liskov, Barbara H.; Snyder, Alan; Atkinson, Russell R.; and Schaffert, J. Craig. "Abstraction Mechanisms in CLU." Communications of the ACM, Vol. 20 No. 8 (August 1977), 564-576.

18. McCarthy, John, et al. LISP 1.5 Programmer's Manual. Cambridge, Ma.: M.I.T. Press, 1962.

19. McCarthy, John. "A Basis for a Mathematical Theory of Computation." Computer Programming and Formal Systems. Edited by Braffort and Hirschberg. Amsterdam: North-Holland Publishing Co., 1963.

20. Parnas, David L. "A Technique for Software Module Specification with Examples." Communications of the ACM, Vol. 15 No. 5 (May 1972), 330-336.

21. Parnas, David L., and Handzel, G. More on Specification Techniques for Software Modules. Fachbereich Informatik, Technische Hochschule Darmstadt, Darmstadt, West Germany, 1975.

22. Robinson, Lawrence; Levitt, Karl; Neumann, Peter; and Saxena, Ashok R. "On Attaining Reliable Software for a Secure Operating System." SIGPLAN Notices, Vol. 10 No. 6 (June 1975), 267-284.

23. Scheifler, Robert W. An Analysis of Inline Substitution for the CLU Programming Language. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 139-1. Cambridge, Ma., April 1977.

24. Sturgis, Howard E. A Postmortem for a Time-Sharing System. Xerox Research Center, CSL 74-1. Palo Alto, Ca., 1974.

25. Sites, Richard L. Proving That Computer Programs Terminate Cleanly. Stanford University, Computer Science Department, CS-74-418. Stanford, Ca., 1974.

26. Wulf, William A.; Cohen, Ellis; Corwin, W.; Jones, Anita K.; Levin, Roy; Pierson, C.; and Pollack, R. "HYDRA: The Kernel of a Multiprocessing Operating System." Communications of the ACM, Vol. 17 No. 6 (June 1974), 337-345.

27. Wulf, William A.; Johnsson, Richard K.; Weinstock, Charles B.; Hobbs, Steven O.; and Geschke, Charles M. The Design of an Optimizing Compiler. New York: American Elsevier Publishing Co., 1975.

28. Wulf, William A.; London, Ralph; and Shaw, Mary. "An Introduction to the Construction and Verification of Alphard Programs. IEEE Transactions on Software Engineering, Vol. SE-2 No. 4 (December 1976), 253-265.

29. Zilles, Stephen N. "Algebraic Specification of Data Types." Progress Report XI. M.I.T., Laboratory for Computer Science. LCS/PR-XI. Cambridge, Ma., 1974.

Publications

1. Liskov, Barbara H. An Introduction to CLU. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 136. Cambridge, Ma., February 1976. Also New Directions in Algorithmic Languages. Edited by S. A. Schuman. Rocquencourt, France: IRIA, 1976.

2. Jones, Anita K, and Liskov, Barbara H. An Access Control Facility for Programming Languages. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 137. Cambridge, Ma., April 1976.

3. Jones, Anita K., and Liskov, Barbara H. "A Language Extension for Controlling Access to Shared Data." IEEE Transactions on Software Engineering, Vol. SE-2 No. 4 (December 1976), 277-285.

4. Scheifler, Robert W. An Analysis of Inline Substitution for a Structured Programming Language. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 139. Cambridge, Ma., June 1976.

Accepted for Publication

1. Liskov, Barbara H., and Berzins, Valdis. "An Appraisal of Program Specifications." The Impact of Research on Software Technology. Edited by Peter Wegner. Cambridge, Ma.: MIT Press, to appear.

2. Liskov, Barbara H., and Zilles, Stephen N. "An Introduction to Formal Specifications of Data Abstractions." Current Trends in Programming Methodology. Vol. 1. Edited by Raymond Yeh. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., to appear.

Theses Completed

1. Atkinson, Russell R. "Optimization Techniques for a Structured Programming Language." unpublished S. M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

2. Fulton, Gordon L. "A Microprogrammed Instruction Set for a 32-bit Minicomputer." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

3. Gorgen, David P. "An Algorithm to Determine Mutability of Data Types in CLU." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

4. McCabe, Edward J. "A Compactifying Garbage Collection Algorithm for a Typed Programming Language." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

5. Scheifler, Robert W. "An Analysis of Inline Substitution for the CLU Programming Language." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

6. Virgile, Kenneth. "MEIL: A Macro Expandable Intermediate Language." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1976.

Theses in Progress

1. Berzins, Valdis. "Abstract Model Specification for Data Abstractions." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1978.

2. Gugenheim, Michael R. "An Inline Substitution Package for the CLU Language." S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

3. Kapur, Deepak. "Towards a Theory of Data Abstractions." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1978.

4. Moss, J. Eliot. "An Abstract Data Type Facility for a Programming Language." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1978.

5. Principato, Robert. "A Formalization of the Parnas Module Specification Technique." E.E. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1978.

6. Schaffert, J. Craig. "A Formal Definition of the Programming Language CLU." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1978.

7. Zilles, Stephen N. "Data Algebra: A Specification Technique for Data Structures." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1978.

Talks

1. Berzins, Valdis. "An Appraisal of Program Specifications." Symposium on the Impact of Research on Software Technology, Durham, North Carolina, July 1976.

2. Liskov, Barbara H. "Data Abstractions." Industrial Liaison Office Seminar, M.I.T., Cambridge, Mass., March 1976.

3. Liskov, Barbara H. "Data Bases and Data Abstraction." Conference on Data, Salt Lake City, Utah, March 1976.

4. Liskov, Barbara H. "Data Abstractions in CLU." Italian Computer Society Meeting, Pisa, Italy, May 1976.

5. Liskov, Barbara H. "Abstract Model Specifications of Data Abstractions." IFIP Working Group 5.2, France, May 1976; Italian Computer Society Meeting, Pisa, Italy, May 1976.

6. Liskov, Barbara H. "Reliable Software." Panel Member, National Computer Conference, New York, N. Y., June 1976.

7. Liskov, Barbara H. "Data Abstractions." Discussant, Tinman Workshop, Cornell University, Ithaca, N. Y., September 1976.

8. Liskov, Barbara H. "A Language Extension for Controlling Access to Shared Data." 2nd Annual Conference on Software Engineering, San Francisco, Ca., October 1976.

9. Liskov, Barbara H. Session Chairman, 2nd Annual Conference on Software Engineering, San Francisco, Ca., October 1976.

10. Liskov, Barbara H. "Data Abstraction and Software Reliability." Control Data Corp., St. Paul, Minn., October 1976; Codex Corp., Watertown, Mass., December 1976; Sylvania - GTE, Needham, Mass., December 1976.

11. Liskov, Barbara H. "An Introduction to CLU." Tufts University, November 1976.

PROGRAMMING TECHNOLOGY

Academic Staff

A. Vezza, Group Leader

J. C. R. Licklider

Research Staff

E. R. Banks
J. M. Berez
E. H. Black
M. S. Blank
M. F. Brescia

M. S. Broos
S. W. Galley
J. F. Haverty
P. D. Lebling
C. L. Reeve

Graduate Students

T. A. Anderson
S. E. Cutler

B. K. Daniels
G. D. McGath

Undergraduate Students

B. T. Berkowitz
D. L. Dill
A. G. Jaffer
T. J. Platt

D. Sherry
S. H. Soto
W. W. St. Clair
T. To

Support Staff

S. P. Briggs

PROGRAMMING TECHNOLOGY

A. INTRODUCTION

The Programming Technology group is engaged in two distinct research and development programs. (1) The program in Morse code has as its main goals the development of the conceptual insight necessary to develop a computerized Morse-code operator and the design and implementation of a prototype of such a computer system (COMCO-I). The Morse-code program covers four areas [1]: signal processing, Morse-code transcription, sender recognition, and understanding of the network conversations among operators that are carried on in a special language consisting of "Q-signs" and "Pro-signs". (2) The other research program is concerned with the facilitation of interpersonal communication through the use of computer message systems. The work on interpersonal communication has involved the design and implementation of a computer message system that embodies in it a model, as yet very simple, of an organization. The model is used to track action status and to aid the communication process.

B. MORSE CODE

At first glance, designing an automated system capable of transcribing a hand-sent Morse-code signal appears too simple to be interesting. For a person, the most difficult aspect of learning Morse code is remembering the pattern of dots and dashes associated with each letter of the alphabet. This type of recall is a simple task for current computer systems. However, experience has shown that human Morse-code operators perform several tasks beyond this mapping of dots and dashes to letters. These tasks are considerably more difficult, and human Morse-code operators perform them far better than current automated systems can.

One such task is locating the signal. In practice Morse-code signals are broadcast over radio waves. A Morse-code operator must be able to tune the receiver dynamically during a session. Should the signal drift, the receiver may need to be tuned to follow it. When signal strength becomes too low for reliable reception and translation, a human operator will recognize this and act appropriately--and so must an automated system. Interference affects reception in a similar manner.

Another difficult task for automated systems is to convert a manual Morse-code signal, consisting of patterns of the five Morse-code elements, into characters and words that the sender sent (or intended to send). Two of the elements, dots and dashes, are called marks. The remaining three are the spaces that separate the marks. Mark spaces separate adjacent marks within a character; character spaces separate adjacent marks belonging to adjacent characters and word spaces separate adjacent marks belonging to adjacent words. Ideally, a dot and a mark space are of equal duration, a dash and a character space three times longer than a dot, and a word space

seven times longer than a dot. Unfortunately, real dashes can be even shorter than a particular dot in the same transmission. The length of a space tends to be even less predictable than the length of a mark. It is interesting that human operators have so little trouble understanding each other's code.

An important observation is that operators tend to have considerably more difficulty transcribing a message in a foreign language than one in their native tongue. In addition, both the sending and receiving operators must be more attentive to their respective tasks if the message is composed of code groups, because the coded message has little syntactic and semantic structure to aid the transcription process. This fact leads to the obvious conclusion that receiving Morse code requires some knowledge about the message. If the message is in English, then each token in the message must be an English word. In addition, the words must follow, in some broad sense, syntactic and semantic English rules. This understanding of the domain, English in this case, is considerably more difficult for current automated systems than for human operators.

During the past year, progress in all of the above four areas has been made with major accomplishments achieved in the signal processing and transcription areas. An event of significance in the signal processing area has been the identification of the need for and the implementation of what traditionally would be considered an "unrealizable" filter (Black, Haverty, St. Clair, Vezza). Equally important have been the improvements to the transcription module COMDEC that provide for the recognition and proper handling of the Morse-code error signs and numbers in clear text (Lebling). Experiments in understanding Morse-code network conversations have led to the realization that the linguistic semantic context alone is not sufficient to understand Morse-code network conversations (Church, Vezza) [2]. In order to understand a Morse-code network conversation, an operator takes into account not only what is being said but who is saying it, even in the circumstance in which the operators on the network do not explicitly identify themselves each time they transmit. Along this line, Anderson [3] has developed a model of Morse-code sender characteristics. He has also pointed out that developing an efficient computer version of the model that would provide for sender fist recognition in real time will be a challenging task.

1. Signal Processing

The most important development in the signal processing area of the Morse-code program was the implementation of a novel tandem phase-lock-loop filter that utilizes time reversal of the input signal; however, the output can be obtained in real time, albeit with a constant delay. The nature of Morse-code signals--the fact that they are on-off keying or frequency-shift keying--and the fact that initial experiments indicated that the transient response of the receiving filter interfered with the measurement of important parameters of the desired signal, led to the need for and the subsequent development of the novel filter. (A software signal-processing module described in

last year's report [1] proved to be unwieldy.)

There is a great deal of information contained in the audio sound of a Morse-code signal--the signal characteristics per se--besides the timing information of the marks and spaces [4] (for a more detailed explanation of the Morse-code project see references 1, 2, 3, 5). It became clear while running some experiments in understanding Morse-code network conversations that the signal characteristics contained information that was an important part of the context of the situation; it was necessary to extract this information in order to understand the network conversations (q.v.).

A small digression is required to explain the kinds of things a human operator hears in the sound quality of the signal and what one must be able to extract from the signal in a computer version. Briefly, a human operator is a very efficient detector, capable of detecting and tracking desirable signals in a crowded spectrum of similar competing but undesirable signals, in a manner almost analogous to the way people follow a particular conversation at a crowded cocktail party. (We say almost analogous, because there is no binaural effect in the Morse-code domain.) This discriminating ability of humans is acute and knowledge-based. To discriminate signals, an operator uses information about how the signal sounds: (a) its frequency; (b) its anticipated frequency drift; (c) its amplitude and rate of chirp, if any; (d) the amount of envelope distortion such as hum, clicks, yoop and whatever other characteristics of the waveform can be characterized. A good signal-processing front end should be capable of measuring some, if not all, of the above signal characteristics and of using the measured characteristics for signal discrimination.

a. A Tandem Phase-lock-loop Filter

The above general requirements can be translated into specific requirements of a receiving filter process for the Morse-code application. (The specific filter design is for an on-off keyed signal, and experiments were conducted only with such a signal. Therefore, the discussion that follows is in the context of an on-off keyed signal. However, it should be pointed out that similar arguments could be made and results obtained for a frequency-shift keyed signal.) Extracting the on-off timing information for marks and spaces as well as signal quality information requires determination of the transitions of the signal as well as continuous estimation of the amplitude and frequency of the signal. The latter information serves a dual purpose. First, it is used to characterize transmitter signals for use in transmitter recognition. In addition, the frequency on which a station is transmitting is part of the situation model, and an uncharacteristic frequency shift of ten or several tens of hertz often indicates a change of "sender" during network conversation. This type of cue is extremely useful, because, after contact has been well established, operators often do not identify themselves.

The tracking-filter model of the phase-locked loop (PLL), Figure 1 [6, 7], is well suited to extracting the information indicated above from a signal in that it gives continuous frequency and amplitude estimates, and presents a relatively narrow-band filter to a frequency-modulated carrier.

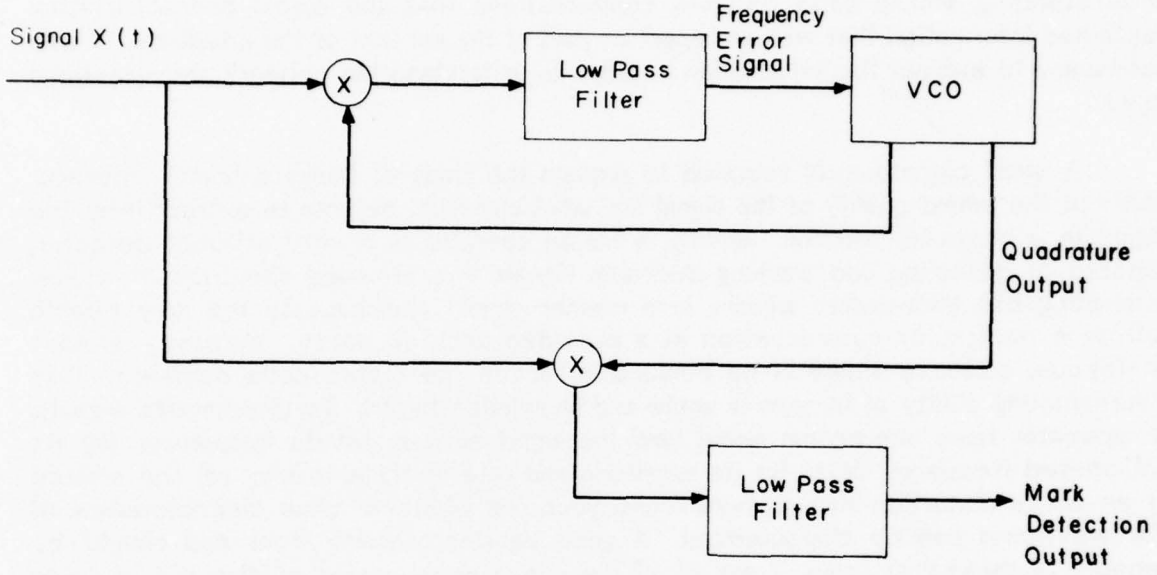

Figure 1. Phase Lock Loop Detector

However, before a PLL can give accurate demodulation, it must achieve lock. (Lock is the state of the PLL when the voltage controlled oscillator (VCO) tracks the incoming signal with a constant phase lag.) The time to achieve lock is inversely proportional to the natural frequency of the loop, and is affected by such factors as the initial frequency error (difference in frequency between the VCO and the input) and noise in the loop.

Chirp is frequency modulation which frequently occurs in low quality transmitters and is often caused by inadequate filtering of the power supply which causes the oscillator to change frequency when the power stage is turned on. Thus, the frequency modulation, or chirp, exists where the signal makes a transition from off to on, or vice versa. Most often, it exists only at the beginning of the "on" period or "mark" of the Morse-code signal. Unfortunately, in a traditional PLL arrangement, or for

that matter any type of traditional filtering, the transient response of the filter is superimposed on the signal and is largest at the signal transition points. The situation is exacerbated when the problem of interference is considered; as one tries to narrow the bandwidth of the filter to eliminate the interfering signals, the period for which the filter transient response is a significant factor in the output is lengthened. In the case of the PLL, the transient between acquisition and lock at the beginning of the signal is the major one, because a PLL will track the signal during the on-to-off transition until it reaches a signal-to-noise ratio at which the signal is lost. Thus, because the frequency and amplitude estimate of the signal prior to lock contains important information, it is desirable to reconstruct that portion of the signal.

A number of ways of recovering the pre-lock information can be conceived. The method settled upon is simple, and we think it is somewhat elegant. It involves sampling and storing the input to the PLL, and then, after the PLL has completed processing the mark in the forward direction, sending the stored samples in reverse order through the loop. The loop then demodulates a time-reversed replica of the original signal, and the original leading-edge information is reliably obtained from the trailing edge of the reversed signal.

Because it was desirable to run the process in real time, only the beginning portion of the signal is reversed and a second PLL is used to demodulate it so that the first PLL can continue to demodulate the forward signal. In addition, the reversed signal can be compressed in time by sending the reverse-order samples through the secondary PLL at a rate faster than they were collected. Of course, the secondary PLL needs to run at a higher frequency. Thus, it is possible to have reconstructed the mark before the next mark begins.

The PLLs are connected as shown in Figure 2.

The input is sampled and stored in a last-in-first-out (LIFO) memory. This portion is currently implemented digitally, in the absence of a cost-effective charge-coupled-device (CCD) solution. Meanwhile the input to the secondary PLL is taken from the quadrature phase of the primary PLL. When lock is indicated (by the quadrature, or correlation output of the primary PLL) the input of the secondary PLL is switched to the digital-to-analog converter (DAC), and the stored samples are read out in reverse order. The outputs of the secondary PLL are taken as the demodulated signal for the time prior to lock.

2. Receiver Control and Signal Acquisition

Using the PLL hardware, a program was written (Haverty) to simulate some of the activity of an operator in searching a segment of the radio spectrum for a particular known signal. Generally, this would correspond to an operator looking for another station with which there is a prearranged schedule to communicate, with the

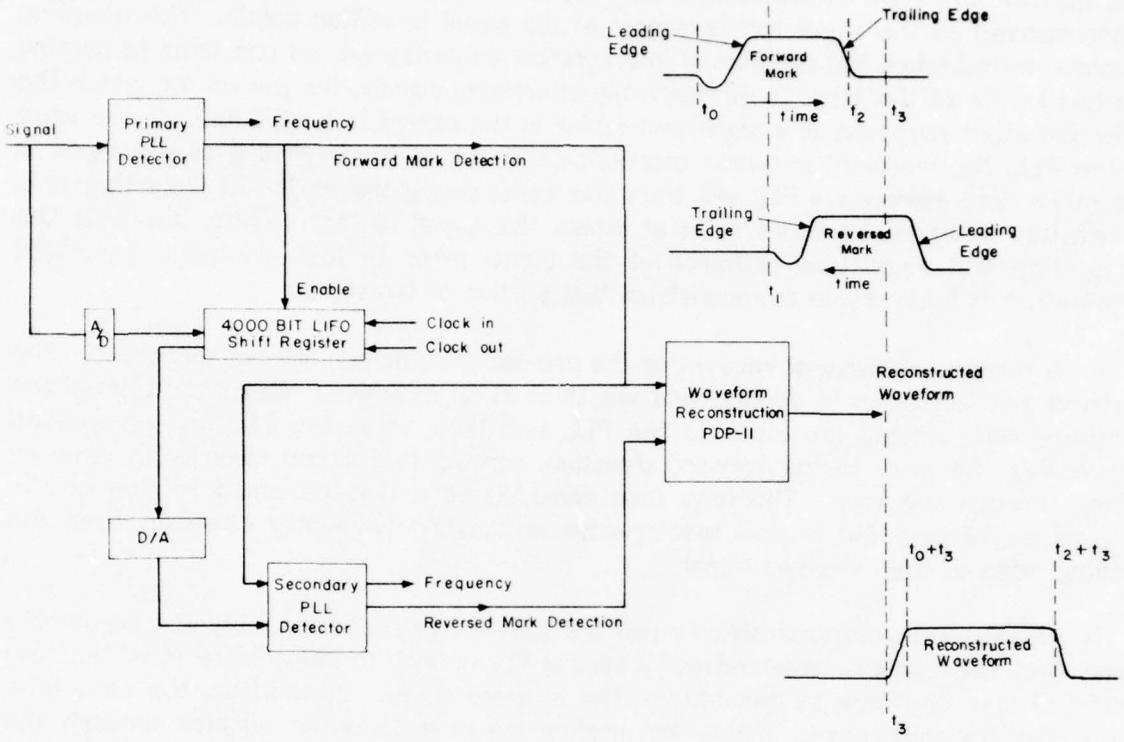

Figure 2. Tandom Phase Lock Loop Detection System with Wave Form Reconstruction

frequency approximately known. The task is to examine the various signals which may be present near that frequency and to find the desired station, at which point the signal detector described above would assume control.

Operators generally use many varied aspects of the signals to assist in identification. The program uses only two of these, which are produced by the PLL, namely the chirp parameters and the signal level. Other parameters--such as the individual senders' code speed and rhythm, radio environment effects such as flutter, language characteristics such as Q-signs used, and so on--are not used, but they could be added as desired to improve the performance of the identification procedure in an integrated transcribing system.

Chirp, if present, is generally a reliable indicator of a particular transmitter, since

it does not change significantly with time, and it is easily measured by the PLL facility. Signal strength, however, is another important clue normally used by operators. It does vary from day to day, but an operator's knowledge of "band conditions" enables one to compensate for these effects to a large extent. The program was therefore constructed to use both of these measures, and compare different samples using a best-fit type of algorithm, to decide whether a signal is "definitely," "possibly," or "definitely not" some particular previously heard transmitter.

The program has been tested using the in-house cable network of transmitters, driven by a standard test tape to simulate several stations simultaneously in operation on slightly different frequencies. The program scans the band segment as directed, determines the number of stations transmitting, and records their frequencies. It is then possible to perform a measurement pass, in which each signal is tuned in and measured in turn, assigned a name, and its characteristic parameter values saved. It is also possible to ask the program to find a particular station by name, in which case it measures each signal present in the segment, compares its parameters to those of the desired signal, and determines which, if any, of the signals match the desired station. Alternatively, a particular signal may be selected for identification by measuring its parameters and determining which, if any, of the signals in the data base it matches.

3. Morse-code Transcription

The capabilities of COMDEC, the Morse-code transcription module, were expanded and improved during the past year (Lebling). The major thrust of its development during the year has been to augment its abilities in translating Morse code into printed text sent in an environment more closely approximating conditions of a live communication between operators. There were several areas in which this effort concentrated, each of which will be discussed in turn:

a. recognition and proper handling of error signs

b. translation of numbers appearing in text

c. English word recognition

d. multiple dictionaries

a. Error signs

Until this year, the transcriber was always run under the assumption that a sender transmits the text straight through, ignoring any known errors. Real Morse-code senders do not in fact behave in this manner.

If a sender makes an error (and notices it!) he or she will resend the erroneous word, phrase, or sentence, signalling with an error sign that this is about to occur. This behavior is somewhat analogous to a typist who spaces back over an error and overstrikes it with X's. An error sign is a sequence of dots sent rapidly, rarely fewer than six, and rarely more than twenty. The number of dots sent varies even within a single transmission, as does the separation of the dot-sequence from the erroneous code preceding it and the "correct" code following it. More importantly, the semantics of an error sign vary even more widely. An error sign may mean to ignore the previous word or it may mean that the previous word or phrase will be resent, and so on. Some examples from actual code (the symbol "@" is used to represent an error sign) follow:

. . . ANY B OY OR GIRL 13 TO 10 @ 19 WHO . . .

The correct translation:

. . . ANY BOY OR GIRL 13 TO 19 WHO . . .

This is the most typical use of an error sign. It signals that the previous word or object was in error, and the sender resends the word correctly. The error sign in this example contained thirteen dots.

. . . PAGE B 23 OF TODAYS PE TPERS @ TODAYS PAPER . . .

This is similar to the previous example, but two words are erased and resent. The error sign contained eleven dots.

. . . THE CORNER OF WASHINGTON BLVD @ AND SCHOOL STREETS . . .

In this example, the word "BLVD" is erased--it should not have been sent at all. The error sign contained seventeen dots.

The problem of error signs therefore is in two parts: recognizing the error sign itself, and finding the area it is intended to erase.

COMDEC finds error signs using a module that is the first to run after the initial Maude-like [4] transcription of the code. This module looks for sequences of five or more dots. When it finds such a sequence, it estimates how likely it is that the sequence is an error sign. Specifically, an ideal error sign should be:

1. composed of nothing but dots and spaces

2. six or more dots long

3. composed of spaces all of the same type, and

4. set off from the surrounding code by word-spaces.

If a dot-sequence satisfies these criteria it is an error sign. If it does not, it may still be an error sign. The most important decision factor in the latter case is how long the sequence is, and how uniform the spaces are. This technique succeeds because few English words (or Morse-code signs) have long sequences of continuous dots in them. For example, the word "THESIS" has thirteen contiguous dots but also has an initial dash and very irregular spacing.

Once COMDEC finds a suspected error sign, it attempts to find the area the error sign erases. This is done by going back in the message, stopping at word (and some letter) spaces. The area of code between the stopping place and the error sign is compared to the code following the error sign. If the code sequences are sufficiently alike, then COMDEC has found the area that needs to be erased. Of course, the area being erased might not be too much like what follows, because it was sent incorrectly. This fact causes COMDEC to give more weight to a correct match with the spacing of the following code, a technique that is similar to a letter-by-letter comparison. A simple example illustrates this problem:

. . . UNDER THE EHEADING @ HEADING . . .

If the "closest match" to the following code were selected as the correct error sign and erasure, then the transcription of this sequence would be "UNDER THE E HEADING." Taking spacing into account, and recognizing that the erased code should contain an error, the correct transcription of "UNDER THE HEADING" is produced.

The message containing the previous examples, and COMDEC's transcription of it, will be given later.

b. Numbers

COMDEC recognizes arbitrarily long sequences of digits as "numbers" (Lebling).

The problem of transcribing numbers is analogous in some ways to that of transcribing error signs, and it arises out of the fact that most of COMDEC's transcribing is vocabulary-based. Since it is theoretically possible to send any number from zero up to numbers containing any number of digits, it is impractical to include them in a "dictionary" of numbers. Instead, COMDEC utilizes the properties of the Morse code used to represent the digits:

1. All digits consist of five marks.

2. All digits contain a sequence of dots followed by a sequence of dashes, or vice versa.

3. A number very often appears in context, that is, as a part of a date, time, address, page number, age specification, and so forth. This context is used to reinforce the possibility of a number. A number appearing out of context must be allowed, as all possible contexts have not been or cannot practically be implemented. If a number appears out of the contexts in which a number is expected, it is looked upon by COMDEC with great suspicion, and it will be allowed to remain a number only if it is well sent compared to other interpretations of what it might be.

COMDEC searches for mark sequences that fit these criteria and then attempts to "expand" them on either side (to produce complete numbers). The only limitation on this algorithm is that at least one digit of an n-digit number must be sent correctly.

c. English Word Recognition

One of the potential problems with a vocabulary-based transcriber such as COMDEC is that it is impossible to have a complete vocabulary. The frequency graph of English is such that after the first few thousand words almost all words are equally frequent (or infrequent). The practical consequence for COMDEC is that any sufficiently long message is likely to contain at least one English word that is not in COMDEC's vocabulary. If a legitimate word is not recognized as such, it can lead to COMDEC's believing that it is a word it knows (or several such words), but one made unrecognizable by a mark error. In the worst case, an unknown word may be "corrected" and split up into several known words.

We have investigated including in COMDEC a module which is able to estimate the likelihood that a given sequence of letters appearing in a message is an unknown word (Sherry, Lebling, Broos). This module is based on the observation that some sequences of vowels or consonants occur in English and others do not. For example, "EA" is a very common vowel sequence, whereas "AA" is very rare. Similarly, some letter sequences occur in the middle of words fairly commonly, but are rare or impossible at the beginning or end. For example, "OO" is common in the middle of words, but rare-to-impossible at the beginning of words.

This word-recognizer is able to recognize over 98% of all nonsense character sequences given it as non-English. It is to be installed in COMDEC during the coming year.

d. Multiple Dictionaries

COMDEC's run-length-sequence (RLS) lookup functions have been improved to allow more than one dictionary to be searched (Lebling). This improvement enables dictionaries for special applications (such as transcribing Morse-code network chatter) to be switched in and out as needed. Eventually such dictionary switching will be signalled by "event markers" (which see) placed in the code.

e. An Example

A major spur to this year's effort was provided by a tape of senders made by several instructors at the Army's Morse-code school at Fort Devens, Massachusetts. The tape contained many different types of code, as sent by trained (but sloppy!) senders.

One section of this tape illustrates many of the problems worked on this year. This section is one of several "messages" being transmitted by the senders on the tape. The message is a transcription of an article which appeared in the Boston Globe at about the time the tape was made.

The correct text, as it appeared in the article, is as follows:

> "In an attempt to alleviate youth unemployment in the city, the Sunday Globe on May 30 will publish free advertisements for Boston teenagers seeking summer jobs. Any boy or girl 13 to 19 who lives in Boston can place a job wanted ad without charge by filling out the coupon on page B 23 of today's paper and mailing it to Summer Jobs, The Boston Globe, Boston, Massachusetts 02107. Teenagers may also take the coupons to the Globe's downtown office, at the corner of Washington and School Streets, or at its main office, 135 Morrissey Blvd, Dorchester. Coupons must be received by 5 pm, Wednesday, May 26. Job seekers may be as specific as they like in mentioning the hours or days they are available for employment, the type of work they desire or can do or what wages they expect. The ads will appear in the May 30 classified section under the heading Hire a Boston Teenager for the Summer."

COMDEC's Maude-like first pass transcribed it as follows (curly brackets indicate uninterpretable mark sequences):

IN ANATT E MPT TOALLEVE EE@HALLE4IATE YOUTV UN E MPLOYMENT I N THE CI TY MIM T @@5 CI TY, T5E SUNDAYGLOBE ON MAY30 WILL PUBLIS5 FRE E ADVERTIS E MENTS FOR 6OSTON T E E NAGERS S E E KING SUMMER JOBS. ANY B OYORGIRL 13TO 10 @@E19 W5OLIVES IN BOSTONCANP LACE AJO6 WANTED AD WI THOUT CAAEEEEEEEEEEEEEEE CH#GE 6YFILLING OU T THE C{---..-}PON ON PAGE B 23 OF

TODAYS PETPER@H@E TODAYSPAPER ANDMAILING I T TO SUMMER JOBS {--..-} T5E BOS{----}N GLOBE, BOSTON, MAS S AC 5 EEEEEEEEEEE MAS SAC5USE T TS 02107. TE E N{.---.}ERS MAY ALSO TAKE T5 E COUPONS T O T5E GLOBES DOWNTOWN OFFICE, A T T5E C ORNER OF WASHINGTON 6LVD EEEEEEEEEEEEEEEEE AND SCH OOL STRE E TS,ORATITS MAI N OF FICE, 13 @E MORRISS E Y BLVD, DORC4 E STER A{.-.-} COUPTANS {--..-}S T B E RE CEIVED {-.....}Y EEEI P M,WEDNESDAY M{..--} MAY{..--} T EEEEEEEEEEE, MAY2T5.JOBSE E KI EEEEEEEEEEE JOBSEEKERS MAY BE &SPECIFICAS TH E Y L IKE I N ME NTIONING T5E 50{..-.-.}S OR DAYS T5EY AR E AVAILABTIIE@I AVAILABLE FOR E MPLOYMENT, THE TYPE OF WTMRK 6 E YDES IRE {---.-.}CANDO{---.-.}WV AT W{.---.} E S TH E Y E XPECT. THE {.--..}SWILLAPPE#INT5E MAY30CLASS IFII EEEEEEEEEEEEEEEET CL&SISE I E D SECTION UNDER EH E EHE {.--..}ING 5I EEEEEEEEE HEADING 5 IRE ABOSTON T E E NAGE RFOR TH E SUMMER.

This is the COMDEC transcription ("@" in square brackets indicates an error sign; "xxxxx" indicates that a portion of the message in error was suppressed from the output; and "<>" indicates a word obtained from the dictionary, assuming the sender made a mark error).

IN AN ATTEMPT TO [xxxxx @] <ALLEVIATE> <YOUTH> UNEMPLOYMENT IN THE [xxxxx @] CITY, <THE> SUNDAY GLOBE ON MAY 30 WILL <PUBLISH> FREE ADVERTISEMENTS FOR <BOSTON> TEENAGERS SEEKING SUMMER JOBS. ANY BOY OR GIRL 13 TO [xxxxx @] 19 <WHO> LIVES IN BOSTON CAN PLACE A <JOB> WANTED AD WITHOUT [xxxxx @] CHARGE <BY> FILLING OUT THE COUPON ON PAGE B 23 OF [xxxxx @] TODAYS PAPER AND MAILING IT TO SUMMER JOBS <,> <THE> BOSTON GLOBE, BOSTON, [xxxxx @] <MASSACHUSETTS> 02107. TEENAGERS MAY ALSO TAKE <THE> COUPONS TO <THE> GLOBES DOWNTOWN OFFICE, AT <THE> CORNER OF WASHINGTON <BLVD> [@] AND SCHOOL STREETS, OR AT ITS MAIN OFFICE, 135 MORRISSEY BLVD, <DORCHESTER>. <COUPONS> MUST BE RECEIVED <BY> 5 PM, WEDNESDAY [xxxxx @], MAY <26>. [xxxxx @] JOB SEEKERS MAY BE AS SPECIFIC AS THEY LIKE IN MENTIONING THE <HOURS> OR DAYS <THEY> ARE [xxxxx @] AVAILABLE FOR EMPLOYMENT, THE TYPE OF WORK <BE> {Y} DESIRE OR CAN DO OR <WHAT> WAGES THEY EXPECT. THE ADS WILL APPEAR IN <THE> MAY 30 [xxxxx @] <CLASSIFIED> SECTION UNDER <SHE> [xxxxx @] HEADING 5 IRE A BOSTON TEENAGER FOR THE SUMMER.

4. Understanding Morse-code Senders

Hand-sent Morse code is like speech and handwriting in that the characteristics of a transmission depend on the sender, in such a way that the sender can often be recognized from the transmission. Although Morse-code transcription systems have historically avoided the problems caused by sender differences, a study of the requirements of a full Morse-code system--one which obtains its input from radio transmissions--shows that information regarding sender differences can be extremely valuable. Morse-code transcription systems have attempted to fit all senders into one

badly-fitting description.

A model of individual Morse-code senders has been proposed (Anderson) [3], and structured to allow its use in a system which attempts to recognize individual senders. The model is based partially on information obtained by averaging over an entire transmission; principally, though, it attempts to describe those structures in a sender's transmissions, such as letters and words, which are sent with consistency. This aspect of the model, although similar in some respects to models used by some transcription systems, removes many restrictions imposed by those earlier models: the structures used are not limited to letters, and data regarding any particular structure may be excluded from a sender's model if it is sent inconsistently. The description of Morse-code senders is seen to include far more than just a description of their "fists;" the model must also contain information useful to the rest of the Morse-code system.

5. Understanding Morse-code Networks

A set of programs for understanding Morse-code networks was implemented and interfaced to COMDEC (Church) [2]. A set of experiments was designed, which hypothesized the existence (or lack thereof) of certain context information from other modules in the system (Church, Vezza). Experiments run on a number of actual Morse-code network conversations produced several interesting results. (1) The syntax of Morse-code conversations, although rather loose, can and does provide useful feedback to the transcription process in order to correct translation of poorly sent Morse-code sequences. (2) Sender transmissions are extremely important as context information. (3) Semantic feedback to the transcription process is currently limited to flagging that which obviously doesn't make sense, but there is no mechanism for correcting the Morse-code sequence. (4) The coupling between COMDEC and the understanding module needs to be tighter and integrated into a more consistent whole.

Morse-code operators have mental models of how people send, what their transmitters sound like, who the members of a particular network are, which members are currently active in the network and where they are in the spectrum. All but the last two models require long-term memory, that is, they are remembered from one session to the next; the last two require short-term memory, as they change from session to session and even during a session. (Even in a simplex network, the various members are likely to be separated by 10 or 20 hertz, and this is enough of a separation at audio frequencies to determine when a sender changes.) These models are extremely important in aiding a person or a system in setting the network contexts, that is, helping identify senders and determining when a sender changes. This context information is necessary, as the linguistic context of a particular Morse-code sequence is often ambiguous.

Consider the example shown below. In italics at the left of each line of text is the speaker, either the network controller (*NCS*, whose call sign is W1HVW), or a

member of the network (*WGI*, whose call sign is WA1WGI). Each line from the conversation is followed by a free translation of the line into English. All call signs are fictitious.

NCS: RN RN DE W1HVW QNI K

The network controller gives his call sign, and asks stations to log in to the net.

WGI: DE WA1WGI QNI QTC 3 BOS

WGI logs in to the net, and reports that he has three messages to be transmitted to Boston.

NCS: QSP SPFD

The net controller, by way of acknowledgement, asks if WGI can relay messages to Springfield.

WGI: C

WGI answers affirmatively.

NCS: DN 5 K1JRW

The net controller asks WGI to go down five kilohertz in frequency, where he should exchange traffic with K1JRW.

WGI: C

WGI acknowledges the transmission and says that he is indeed going down five kilohertz.

The preceding dialogue can be interpreted differently, if we assume that WGI's first transmission ends slightly later in the message.

NCS: RN RN DE W1HVW QNI K

This line is exactly as in the first dialogue.

WGI: DE WA1WGI QNI QTC 3 BOS QSP SPFD

The ambiguity is introduced at this point: WGI again logs in to the net and reports that he has traffic for Boston; this time, though, he also asks whether the net can relay traffic to Springfield. The meaning of QSP SPFD has not changed; rather, the object of the question it asks has changed. A transcript of the dialogue without speaker transitions would not show any difference between the first and second dialogues.

NCS: C DN 5 K1JRW

The network controller answers that the net can relay messages to

Springfield; he then dispatches WGI off frequency to exchange traffic with K1JRW, as before.

WGI: C

As before, WGI acknowledges.

Thus there are at least two acceptable interpretations of the dialogue, depending on the locations of speaker transitions. There is still another interpretation of the second dialogue, depending on the global context: if the net controller had been looking for someone to relay messages to Springfield, WGI's QSP SPFD might mean "Yes, I can relay messages there." Thus, in addition to the speaker transitions, a program attempting to understand this dialogue would have to know what had gone before.

C. INTERPERSONAL COMMUNICATION

The program in interpersonal communication has centered about the design and implementation of a Data-based Message Service (DMS) (Broos, Berez, Blank, Brescia, Galley, McGath, Platt, Vezza) [8, 9]. It is "data-based" because the messages it manages are data in a relational data base.

The central design principle in DMS is that a message service is (or should be) data-base intensive. By that we mean that an on-line data base may contain thousands or even tens or hundreds of thousands of messages. The data base must be capable of being updated frequently as new messages arrive and as users annotate existing messages or specify their own idiosyncratic filing keywords. Further, the user needs the capability for finding and retrieving a message or group of messages in an easy, natural, and computationally efficient manner. In addition, data-base intensive systems like DMS can be naturally integrated with management information systems. For storage efficiency, parts of the data base may be shared among many users while other parts remain private to the individual. For example, all of the recipients of a message may share the text of that message, but annotations they may make to it can remain private.

The general model on which DMS is designed is that of a typical office. Superimposed on this office model is a simple but specific model of a Naval organization. The interface at an intelligent terminal between DMS and a user is designed to be comfortable and familiar to people not used to working with computers. Concepts and terminology from typical office methods of managing paper-based messages (letters, memos, and so on) are used wherever possible, rather than computer terminology. The interface is also designed to be robust and resilient, in the sense that there should be nothing that a user can do that will cause the system to take an irreversible action that the user will regret, without giving the user an

opportunity to reject the action. It is important that this opportunity to reject actions not hinder the user's intended actions. For instance, asking a user to confirm an action unnecessarily, such as a deletion, is a hindrance of his or her intended action.

Message systems like the one described here are really the beginning of "office automation systems," because they are more than just simple message creation and delivery systems. Such systems must possess user interfaces which are reasonable and easy for people to use, and that give a user the feeling that she or he is definitely in control of the machine, rather than vice versa. For example, DMS is largely "form-driven," in the sense that the user creates and changes messages and other objects by filling in forms, rather than answering a sequence of questions or, worse yet, having to remember the order and meaning of parameters in a command. Another DMS design principle is that the computer should sound like a mechanical servant rather than some sort of person; sentences directed at the user should be phrased "That can't be done" rather than "I can't do that."

1. Configuration

DMS operates on a hardware configuration that includes:

a. a central PDP-10 computer that contains the data base and serves all DMS users at the installation

b. a number of intelligent terminals connected to the central computer, each with a cathode-ray-tube display and both a typewriter-like keyboard and special function keys, designed to make DMS easily and naturally accessible to users

c. a smaller number of high-speed printers connected to the central computer, preferably located so that every terminal is reasonably close to a printer, to provide users with paper copies of messages when that is required; and

d. a telecommunications facility that interfaces the central computer with communications lines or a computer network, so that DMS can receive messages from and transmit them to computer-based message services at other sites.

DMS currently supports Hewlett-Packard 2649A terminals, which contain a micro-processor and video display. DMS makes extensive use of this terminal and of a program for the terminal's micro-processor developed at the Information Sciences Institute of the University of Southern California [10].

A DMS installation consists of the following software, operating under the auspices of a general-purpose operating system (currently Tenex only) and programmed almost entirely in the structured Lisp-like language MDL [11] and the file system ASYLUM:

a. a central relational data base for information shared by all users

b. a smaller relational data base for each user's individual information

c. a process for each user, containing components including a command parser that interprets the user's commands, a "message vault" that provides access to the data base, a "virtual terminal" that interfaces to and complements the capabilities of the user's terminal, and programs to perform each kind of command; and

d. a number of processes that may run in the background, performing computation-intensive tasks such as local message distribution, remote message transmission and reception, index updating, message formatting, etc.

Because MDL is a structured programming language, new capabilities for any of these processes are fairly easy to implement and install, if and when they are needed. Background processes are especially flexible, because there is no need to ensure that each user is using the newer program. The "virtual terminal" concept of DMS ensures that all the necessary functions can be provided, by either the terminal itself or the central computer. The parts of the central-computer program that support the virtual terminal make a separate module, so that potentially several different kinds of hardware terminals can be used with DMS.

2. The Data Base

The major characteristic of DMS that distinguishes it from other message systems currently under development [12, 13] is that it is built on top of a data-base system rather than a text-processing system. DMS was designed this way so that it can be used as effectively with large numbers of messages (say, tens or hundreds of thousands) as with the relatively few messages used in testing environments.

A DMS data base is organized on the relational model [14], in which the information is stored conceptually as a two-dimensional array. All messages are stored by DMS at a central computer installation as rows ("tuples") in a single, potentially large relation. The relation is "un-normalized," in the sense that a field ("column") of a message can contain more than one value (data element). The messages in the relation are not ordered, except by identification number; access to the relation is through indexes.

The word index is used here in a sense much like an index in a book: an index is an ordered list of all values occurring in a particular field of the relation (analogous to a list of important words in a book), and associated with each value in the index is a list of message numbers in which that value occurs (analogous to a list of page numbers in a book's index). Actually the field values are organized not in a list but in

a tree, so that the DMS command parser can locate them quickly. Similarly, the message numbers are organized not in a list but in a combination of lists and bit-masks, for storage and update efficiency [8]. Thus there is a nearly constant cost for retrieving messages from the relation according to selection criteria that involve indexed fields. The manager of a DMS installation can designate (for each user) which fields are to have indexes. Non-indexed fields must be searched linearly; that is, the field values in each message must be individually examined, making the cost of searching rise linearly with the number of messages involved. But the cost is minimized for retrieving messages according to conjunctive ("ANDed") criteria that involve both indexed and non-indexed fields, because a search-optimizing module causes indexed searches to be performed first, so that linear searches are performed on only those messages meeting the indexed-field criteria.

The basic trade-off in an indexed (or, more typically, partially-indexed) data base is between the amount of time spent maintaining the indexes and the decrease in retrieval time that such indexes make possible. Fortunately, the data base used by DMS is organized in such a way that updates do not require complete reorganization of the data base. Even in a large data base, insertion of a single new message, along with maintenance of the associated indexes, will result in the modification of only a small fraction of that data base's disk pages. The two main reasons why this is true have been mentioned before: namely, the relation of messages is not ordered, and the indexes are data structures, not simple ordered lists of message numbers.

There are three kinds of message fields in DMS: external, organizational, and personal. External fields are those that are received from or transmitted to outside the organization, for example, address, subject, and text. Organizational fields are accessible only within DMS, by any user that has access to the message; for example, notes, retrieval keywords, approval lists, responsibility lists, and other information that is to be seen only within the organization. Personal fields have personal values, that is, each user has values that are accessible only to himself or herself, for example, private notes and keywords.

Corresponding to these three kinds of fields are three areas of storage for field values. The central storage area contains values for external fields. An organizational storage area contains values for organizational fields. (Potentially, more than one organization could share use of a single DMS installation without conflict or compromising privacy. In this case there would be an organizational storage area for each organization using the installation.) Finally, each user has a personal storage area containing her or his values for personal fields.

In a sense, this division of storage is invisible to users, because each user sees a message as a whole, with field values taken from the appropriate storage area and merged for the terminal or printer. The storage method means that only one physical copy of a field value needs to be kept, no matter how many users have access to it.

However if organizational policy dictates that users must pay for using DMS, then users would typically pay more for a larger amount of information in their personal storage areas. If a special administrative program were run to expunge old messages from the central data base, then typically users to whom those messages were still accessible would find the messages moved to their personal storage areas, where those users would have to bear the cost of retaining the messages on-line.

To illustrate use of secondary storage: an experimental DMS data base contained 207 messages, with an average length of 855 computer words or about 4200 characters. On a proportional scale, the secondary-storage space used by an average message and its adjunct data was as follows:

1.00	original bare message
1.52	parsed and structured message
1.65	formatted message
0.73	formatted message header
0.11	formatted message summary

5.00	total storage used

The formatted versions are designed for display on a user's terminal. If space were at a premium, the original bare message could be erased after it is parsed and structured, and the formatted versions could be erased at the expense of the time needed to format the message each time it is displayed.

Another aspect of the three kinds of fields is to what extent their values can be changed. The following rules achieve concurrency control and the assurance that external messages (transmitted or received) cannot be altered. An "unshared" message, that is, one that was created by a DMS user and never sent, released, or added to a shared folder (see below), can have its values changed any way the creator desires. All other messages can have their values changed only in accordance with what kind of field is involved. The values of external fields are inviolate in the sense that no user can change their values. This rule means that, once a draft message has been coordinated with colleagues for suggested changes or approval, the desired changes in external fields are not made in the selfsame message; rather a new message is created (by DMS), and the desired changes are made before the new message is seen by anyone other than the drafter. The values of organizational fields can be changed only by appending new values, so that no problem occurs if more than one user changes the same field of the same message concurrently. Values of personal fields can be changed freely.

To meet the demands of DMS data management, two packages of functions were created (Blank), as generalizations of previous software [1], to:

a. allow MDL objects to be created and destroyed outside the normal heap area in primary storage, so that the MDL garbage collector can save time by ignoring them (each such object incurs six computer words of overhead)

b. allow these special MDL objects to be written to and read from secondary storage directly, without modifying addresses or using temporary storage

c. organize secondary storage in a way that avoids the limitations typically set by available PDP-10 operating systems and encountered by builders of data-base systems.

This last facility (named ASYLUM) is in effect the logical-organization part of a file system, with physical-storage management left to the operating system. The file-system is designed for data bases, and it provides:

a. an escape from directory-size limits, since up to 14 million files can be stored in one file directory, rather than the two hundred or fewer allowed by ITS or Tenex

b. an escape from page-size waste, since space for files is allocated in units of single storage words, rather than pages that are hundreds of words in size (the directory overhead for a file is slightly more than four computer words)

c. any number of locks for reading and one lock for updating a file, all concurrently; and

d. identification of files by either name or number: file numbers are used throughout DMS for faster execution.

All four of these features together, in one file system, make it extremely useful for data-base management.

3. The User's View

A DMS terminal has a display screen, typewriter keyboard, function keys, and local processor and storage. The display screen is divided by DMS into three independent windows, in which the user respectively enters commands, examines information generated from command execution, and drafts new messages. What the user sees in each window is a few contiguous lines in a potentially large "page" full of lines. The user can "scroll" a window to bring different lines into view, one at a time. The user can copy information from one window to another, for example, to copy an address from an old message either into a new message being drafted or into a command to search for other messages with that address.

The user creates and edits messages by filling out a form that has a label and blank space for each field of interest. Pro forma messages can be stored for later use, when they can be edited (if necessary) and sent. The user enters commands to DMS in statements that resemble restricted English, for example, "Show messages from Smith." The command parser (Brescia) is "friendly" in that it allows abbreviations; if an abbreviation is so short as to be ambiguous, the parser repeats the command back to the user, expanding abbreviations as best it can, and places the terminal's editing cursor at the exact point of difficulty. In most cases, the user can fix the command with one or two keystrokes and re-enter it. Or the user may not know how to fix the ambiguity. For instance, if one is searching for a message for which the author's name is not precisely known, a request to obtain all author names in the data base that begin with "Stein" is easily fulfilled. Thus, an operation analogous to scanning down index tabs in a file cabinet is provided.

The three windows on the terminal screen are named: (1) the Command Window, where commands to DMS are entered (2) the Information Window, where messages and other information that is only to be viewed is displayed (3) the Draft/Edit Window, where new messages and other objects are created or edited. (In addition, there is a "flash window," where one-line terse comments from DMS are flashed to the user; it has none of the flexibility of the other windows.) The user or DMS can increase the size of any one of the three windows at the expense of the other two windows. Thus the display is always composed of two windows, each displaying two lines of text, and one window displaying 16 lines of text. (The flash window and a "name line" for each of the three windows complete the total of 24 lines.)

There are three kinds of text that can appear on the terminal. The first kind is called editable, because the user can edit or modify it by placing the cursor at the point where a modification is to be made. Most of the text in the Draft/Edit Window and the last line in the Command Window is of this kind. A second kind, called enterable, is not editable, but the cursor can be moved into it for purposes of indicating to DMS the object(s) on which action is to be taken. All of the text in the Information Window is of this kind. The third kind is called non-enterable, since the cursor will jump over it. This kind of text is used whenever labels like those in a form are displayed.

Not only do the function keys provide a means for directly entering certain frequently-used unvarying commands, but also they provide means for controlling window size, what is displayed in each window, and deletion and copying of information within and between windows.

A user's view of the data base is closely analogous to the way messages (letters, memos, and so on) on paper are stored in a typical office. All of the following properties of messages (bins, tags, and folders) are implemented in the same way that

indexes for fields are, and they provide "handles" for specifying sets of messages in the same way. The user can apply any DMS command to any set of messages specifiable by a field/value condition, or by one of the following "handles," or by an arbitrary Boolean combination of these. If the user is unsure how to specify the set exactly, "browsing" commands can be used to "home in" on the set in steps.

Each message accessible to the user has a conceptual location, either in one's file cabinet or on one's desk (called the workspace). The workspace is further divided into four parts: the in-box, where DMS puts new messages addressed to the user; the pending bin, where the user can put messages that need further action; the draft bin, where new messages being drafted are kept; and the discarded bin or wastebasket, where messages can be put to be destroyed by the user or by a janitor process. These conceptual locations are not inherent in the data base; rather they aid the user's mental model of the data base. They also provide some computational efficiency. For example, just as an office worker might look on his or her desk for an object of interest, the DMS user can direct the system to look in the Workspace for a message of interest, rather than having DMS always search the entire data base.

Any message can have one or more tags conceptually attached to it, analogous to the little colored metal tags (called "signals" in the trade) that can be attached to the edge of a piece of paper. DMS tags are automatically added to or removed from DMS messages, and each indicates to the user some notable property, for example, "you have not yet seen the text of this message," "you have not yet seen any part of this message," "this message was delivered to you since you began this session," "this message needs action by you," and so on. Tags are, of course, completely independent of the message's conceptual location.

Another, independent way the user can organize messages in the data base is to group them into folders, analogous to the manila folders typically used to group paper messages in an office file cabinet. However:

a. A message need not be in the file cabinet to be in a folder. In fact a folder can contain some messages in the file cabinet, some in the in-box, some in the draft bin, and so on.

b. A (citation to a) message can be in any number of folders concurrently.

c. The owner of a folder can grant three kinds of access to other DMS users: seeing messages in it, adding messages to it, and removing messages from it.

In combination, these properties of folders allow great power and flexibility in organizing messages in meaningful ways. For example, a folder can contain a draft message plus the message to which it is a reply (and other messages for background information), the folder can be shared with the users that need to approve the

message, and suggested revisions can be added to the folder as they arise.

Each message in the data base is uniquely identified by its control number, a positive integer that is assigned to the message when it is first stored in the data base. All messages have control numbers, both informal messages that go from one DMS user to another and never leave the data base, and formal messages that are received from or transmitted to outside DMS and which represent official communications for which the organization is responsible or accountable. Control numbers can range up to 14 million, limited only by the ASYLUM file directory. A message can always be specified by its control number, should the preceding "handles" seem to be inadequate.

A user can see a message in a number of formats, which specify which fields are to appear on the terminal display screen or printer paper, and where (McGath). Formats are specified in an English-like language by the DMS installation manager. For example, the "full" format typically shows the entire message; the "summary" format shows a few fields of the message on one or two lines, to give a quick idea of the content of the message; and the "action" format shows the action status of the message on one line.

4. Office Model

A person is registered as a DMS user in a special table in the data base containing a unique name for the person and a unique password, known only to the person and to DMS. As part of its simple model of an office, DMS recognizes that a person can "wear different hats," that is, assume different organizational roles, at different times. Thus roles are also registered in the data base, along with a list of which people are allowed to assume each role. A person can assume a role (if desired) at the beginning of an operating session with DMS, and DMS will refer to that role's data base instead of her or his personal data base.

One example of a role is that of shift supervisor in a plant which operates around the clock. During each shift, a different person is expected to assume the role of shift supervisor. Messages concerning the operation of the plant are normally sent to the shift supervisor, to be acted upon by whoever is currently assuming that role. If, instead, a message were sent, by name, to the actual person expected to be assuming the role, it might not be acted upon if that person is absent and replaced by an alternate, or if the shift terminates before the message reaches him or her.

A distinction is carefully kept between the parts of messages that are used only within DMS and those that go "out the door" through the telecommunications facility. Provision is made for the typical office procedure of allowing one user to draft a message, circulating it among colleagues for suggested changes or approval, and requiring a different user, such as a superior, to actually release it as an official

organizational communique (this is analogous to signatory power). While a message circulates locally, the organizational fields are used to pass information about the message (for example, annotations and approvals) among DMS users. A message can be sent to local users freely, but only certain users working at certain terminals are authorized to release a message for transmission outside DMS, and even then a user must confirm the command if the message has not been given all the approvals it needs.

Another concept built into DMS is that of action. According to this model, a message received from outside may put an obligation on the organization to act or respond in some way. The obligation is given to a particular user, either automatically by the reception process or manually by an "incoming-message distributor," who is another user (or both, in that order). This obligated user is the action assignee of the message. The action assignee can "pass the buck" to another user (typically a subordinate) by assigning action again, and so on, until some user declares that action has been completed, that is, the obligation has been fulfilled. DMS helps users keep abreast of these obligations using tags on the messages and special ways of formatting messages to see the action status. (By design, DMS has no way to check up on a user who claims that action on a message is complete; that task is left up to management policy.)

5. Security

Access to the data base is governed by strict security rules. Each value of each field of each message in the data base has an associated security level, one of four possible security levels. This security model is the one used for general military messages. The granularity for security classification is as small as practicable, much smaller than that currently available in computer systems, and it allows separate security levels to be assigned to small units like individual paragraphs in the text. In addition each message has an overall security classification. While this is a military model of security, the same security scheme can also be useful in a civilian installation, where the security levels can be "proprietary," "company confidential," etc.

The security level of field values is indicated in the terminal's Information Window by off-screen lights and in the other windows by highlighted security tokens. Each token is one or two letters, and it indicates the security level of all information following it, up to the next token. The user can change a token or insert a new one, using function keys, to change the security level of any desired information. (Lowering the level requires the user to view all the information and then confirm the change.) Overall, security levels are sufficiently prominent without being obtrusive or hindering to the user.

Each user's view of the data base of messages is first of all limited by the operating security level, declared at the beginning of a session. (Each user and each

terminal has a maximum security level, and DMS will not allow a higher level to be used.) The user can only see field values that have a security level at or below the operating security level, in messages whose overall security level is also at or below the operating security level. Within that restriction, a user's visible data base consists of all messages that are addressed to or created by him or her, plus all messages in folders that are accessible to him or her. To provide a security audit trail, each message includes a list of all users that have ever had access to the message.

One of the goals of the DMS project has been the identification of the security-related primitives which are required to support true multi-level security in transaction-based systems, with a view towards the incorporation of such primitives in future operating systems destined for installation in secure sites. The rationale behind such a "kernel" approach is that, by localizing the security tests in one small area of the operating system (the "security kernel"), one facilitates the necessary task of system verification. Moreover, once verified, the operating system provides an environment in which any number of application programs, which do not themselves need to be verified, can be developed, tested and run. Such application programs can be modified as the user requirements change, without having to undergo the expensive and time-consuming verification process before release of each revision.

The underlying principle of the security kernel is the maintenance of security in programs through control of those programs' input and output (I/O). Application programs on time-sharing systems typically do not manipulate I/O devices directly; rather they rely on the operating system to mediate for them. The operating system thus manages a scarce resource, facilitating its use and protecting users from one another. In a secure operating system, the management of I/O is contained within the security kernel. An active process in the operating system has an associated security level. If a process attempts to read data from an I/O device, the kernel will allow it to read only data which is at or below that process's security level. Data being written to an I/O device is always treated by the kernel as being at the same security level as the process which is writing it.

In the DMS implementation, the security kernel is simulated by a kernel in the application program itself. The kernel is divided into two parts, containing primitives to handle the two I/O devices used by DMS, namely, secondary storage (disk and printer queues) and an intelligent display terminal. The secondary storage kernel, called the "message vault," allows the application program to create and access two-dimensional arrays (messages). Each row (message field) of an array may contain any number of columns (field values), each value (paragraph, word, etc.) having its own security level. The vault primitives allow processes to access data by specifying array number, row number, and column number. A process cannot access a value whose security level is above that of the process, and any values created by a process inherit that process's security level.

AD-A061 932 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
LABORATORY FOR COMPUTER SCIENCE (FORMERLY PROJECT MAC) PROGRESS--ETC(U)
OCT 78 M L DERTOUZOS N00014-75-C-0661
UNCLASSIFIED LCS-PR-14 NL

2 OF 2
AD
A061 932

END
DATE FILMED
2 -- 79
DDC


F
932

There is also an overall security level associated with each array. The kernel will allow a process to access an array only if the security level of the user for which that process is acting is at or above the overall security level of the array. For example, a process acting for an "unclassified" user is denied access to any part of a "confidential" array, even to those values which are "unclassified."

To sum up, then, there are four different security levels which are used by the kernel: the level of the user, which remains constant throughout a DMS session; the level of the process acting for the user, which level may change during the session; the overall level of each array in the vault and the level associated with each value in each array in the vault.

The second part of the kernel mediates access to data stored in the intelligent display terminal, which is communicating with the user. Access to the terminal data is structured along the same lines as access to vault data, and the same security rules are enforced by the terminal kernel. The security level of individual values is indicated on the terminal's screen by highlighted security tokens. The user can, by interacting with the terminal kernel, modify both the contents and the security levels of displayed values, as well as create new values and assign security levels to them. There may be several distinct arrays displayed on different areas (called windows) of the terminal's screen at one time.

The design of the DMS security kernel convinced us that an additional facility was required in the kernel in order to allow some of the most powerful capabilities of DMS to be realized. In short, the kernel allows a process, which is operating at one security level, to call a subroutine which, through the kernel's mediation, is run at a lower level than that of the process calling it. Essentially, the kernel maintains a separate page map for each security level allowed to the user. When a process calls a lower-level subroutine, the kernel substitutes the page map for the lower level, copies the subroutine's arguments into those pages, and calls the subroutine. The process is then running at the lower level and can utilize whatever kernel primitives it requires, subject to any restrictions imposed by its current security level. It has no access to any data other than its arguments and whatever it can access through the kernel. Data in higher-level page maps is not accessible at all. When the subroutine returns, the security kernel reinstates the higher-level page map, copies the subroutine's result (if any) into it, and returns control to the process, which is now operating at the higher level again. The only data being passed down to the lower-level subroutine are the arguments to that subroutine. The kernel enforces the restriction that these arguments must be integers, such as array, row, and column numbers.

Given this subroutine facility, it is possible for a "Trojan Horse" process to leak classified information from one security level to a lower one, if there is deliberate collusion between the higher-level process and a lower-level subroutine. However,

we believe the chance is small that a program bug could inadvertently pass information down in this way. The subroutine facility provides for a more natural human-machine interface, but the risks involved in having it need to be studied more carefully and a policy concerning its use developed.

We encountered a second problem concerning security, which we call the "workspace problem." Currently, any user of a computer system must indicate beforehand the security level of the information he or she is about to input. Failure to indicate a sufficiently high security level can result in an automatic breach of security, especially if a "Trojan Horse" process exists. This problem needs further study, and a better analogue between traditional paper workspaces and computer workspaces needs to be developed.

D. OTHER PROJECTS

During this year, the multi-purpose English sentence analyzer/parser was brought to operational state (Banks), allowing its use with the keyword extractor and thereby allowing automatic document or abstract classification [1].

1. English Sentence Parser

The main improvement to the English parser is the ability to analyze much more complex clauses with nested conjunctions and several complement constructions. Conjunction analysis is still a relatively weak area, however. The dictionary was extended by adding more type information for the verbs; for example, a verb might be the type that allows a noun complement but not an adjective: compare "they elected her president" to "he painted the wall blue." There are now about 18000 words in the dictionary, counting all inflected forms as distinct. Many words have several different meanings. The context-based disambiguator is now operational (Dill). Disambiguation is another weak area at present, but the solution seems to be use of case frame information. The main problem is the updating of the dictionary to represent this information for each verb meaning.

2. Keyword Extraction

This project seeks to automatically extract keywords from English language text by using a variety of heuristics--notably by performing a syntactic analysis of the sentence and using the result of that analysis in a high-level key-phrase extractor. (We use the term "keyword" to denote single words, multi-word phrases, and also the result of transforming such keywords into new entities--for example, "blimp" into "aircraft.") As an example, several newspaper articles about the Argo Merchant oil tanker break-up have been analyzed automatically. Currently only the first paragraph or two of each article were used--with a larger sample more keywords will be selected. Following each sample text below are the keywords automatically extracted.

It should be noted that these keywords were automatically selected from a larger set of keywords extracted and that, for different purposes, a different set of keywords could have been chosen. For example, in a high-recall application, the entire set of keywords could be used at the expense of a lower precision. (Recall is that fraction of relevant documents which are retrieved, and precision is the fraction of retrieved documents which are relevant.)

This is a sample paragraph (from the Boston Globe):

"The coast guard yesterday stepped up its preparations for removing the oil tanker Argo Merchant's remaining cargo of no. 6 residual oil, but said that 1.5 million gallons had already leaked into North Atlantic waters. Coast guard officials, who on Saturday had estimated that 140,000 gallons had already leaked from the 18,743-ton, 641-foot tanker, revised their estimates yesterday morning after receiving reports from aboard the ship which ran aground last week 27 miles southeast of Nantucket. Winds and currents thus far have carried the oil away from land."

These are the keywords automatically extracted from this sample: (Upper-case words are represented in the same way in both parser output and dictionary.)

Classifiers: "18,743-ton 641-foot tanker", "coast guard official", "coast guard", "oil tanker", "number 6 residual oil", "atlantic water"
Key nouns: "official", "saturday", "estimate", "ship", "mile", "nantucket", "land", "current", "wind", "wind and current", "water", "gallon", "oil", "cargo", "argo merchant", "tanker", "remove", "preparation", "guard", "report"
Key-noun meanings: OFFICIAL, ESTIMATE, SHIP, LAND, FLUID-CURRENT, WIND, WATER, OIL
Key proper names: "ATLANTIC-OCEAN"
Key verbs: "estimate", "revise", "receive", "run AGROUND", "carry", "leak", "say", "remove", "step up"
Verb-object combinations: "revise estimate", "receive report", "carry oil", "remove tanker", "step up preparation"
Subject-verb combinations: "official estimate", "official revise", "ship run AGROUND", "wind carry", "gallon leak", "guard step up"
Transformations: REPORT-VERB
Generalizations: PERSON, DAY-OF-WEEK, OFFER, TEMPORAL-LOCATION-VALUE, TIME-PERIOD, LENGTH, COMMODITY, GEOGRAPHIC-OBJECT, WEATHER-CONDITION-VALUE, BEVERAGE, PHYSICAL-SUBSTANCE, FUEL, BOAT, INCREASE-VERB, WRITTEN-MATTER
Contexts: GOVERNMENT, PUBLIC-ADMINISTRATION, ADMINISTRATION, TRANSPORTATION, WATER-TRANSPORTATION, WEATHER, GEOGRAPHY
Unknowns: "nantucket", "leak", "argo merchant", "cargo", "gallon"
Quantities: "140000 gallon", "27 mile", "1500000.0 gallon"
Unresolved ambiguities: "remove", "guard"

The above keywords were selected using the current version of keyword extraction heuristics. It should be noted in the above example that the keyword extractor was able to extract "coast guard" and several of the other combination key phrases--even though it had no previous knowledge of the term "coast guard" in its dictionary, and both "coast" and "guard" are in the dictionary with both noun and verb meanings. Also notable is the ability of the system to generalize many of the keywords via a taxonomy: "oil" to "fuel," "ship" to "boat," etc. It is a simple matter to modify some of the combined forms of keywords via such generalization. However, rather than choosing to do this at keyword extraction time (it is a more expensive operation than keyword extraction itself as currently implemented), we instead allow systems which plan to use the keywords to do the generalizations, for example, the document tagger described below. We haven't yet implemented several trivial heuristics that would eliminate such transformations as "official" to "person."

By performing a syntactic analysis of the sentence, various combinations of keywords can be formed. We have shown empirically that these combinations are particularly good for retrieval applications. A basic feature different here from previous work is the level of syntactic and semantic analysis. For example, the SMART system [15] was unable to detect the similarity in "The chief executive visited Brezhnev" and "Brezhnev was visited by the chief executive" even though it would recognize the similarity in "oil removal" and "removal of oil."

Another feature of our system is the disambiguation of various keywords. For example, the word "jar" has quite different meanings in the following two examples:

"The jar contained some money."
"The impact jarred Boston."

The keyword extractor can frequently disambiguate a meaning for such a word, giving "jar" in the sense of "vessel," or in the sense of "shake," etc. To do this, it uses such information as the part of speech required, general context (from one of the keyword modules), and other information.

Here is another sample (from another newspaper):

"The captain of the Argo Merchant, the tanker that ran aground off Nantucket in the early morning of Dec 15, testified yesterday that he was as much as 24 miles from where he thought he was when the ship ran aground. He said the ship was being steered by compass because the more accurate gyro compass had been malfunctioning periodically during the voyage and the day before the grounding had become erratic, showing an error of as much as 6 degrees on either side of the course."

These are the keywords selected from the above sentences:

Classifiers: "gyro compass"
Key nouns: "compass", "voyage and day", "voyage", "erratic", "show", "error", "degree", "course", "ship", "mile", "december", "nantucket", "tanker", "captain"
Key-noun meanings: VOYAGE, COURSE, SHIP, CAPTAIN
Key proper names: "UNKNOWN-PROPER-NAME"
Key verbs: "say", "steer", "malfunction", "ground", "become", "show", "side", "run AGROUND", "be", "think", "testify"
Verb-object combinations: "say steer", "steer ship", "become show", "show side"
Subject-verb combinations: "compass malfunction", "erratic become", "error side", "ship run AGROUND", "captain think", "tanker run AGROUND"
Transformations: VOYAGE-VERB
Generalizations: PHYSICAL-EVENT, BOAT, LENGTH, MONTH, TEMPORAL-LOCATION-VALUE, PERSON
Contexts: WATER-TRANSPORTATION, TRANSPORTATION, LAW
Unknowns: "erratic", "compass", "ground", "malfunction", "nantucket"
Quantities: "6 degree", "24 mile"
Unresolved ambiguities: "degree", "day", "become"

Classifications on this document (see next section):

OIL-SPILL 1.4 0.116

Another sample:

"The lawyer for the company that insured the oil carried by the grounded tanker Argo Merchant revealed his main contention that the owners had been negligent in maintaining the ship to the point that they risked an accident. We have no quarrel with the way the captain or the crew conducted themselves. We are trying to show that the owners were at fault, that they were negligent."

Key nouns: "quarrel", "way", "captain or crew", "captain", "crew", "fault", "accident", "point", "ship", "maintain", "negligent", "owner", "contente", "argo merchant", "tanker", "oil", "company", "lawyer"
Key-noun meanings: CAPTAIN, CULPABILITY, SHIP, OIL
Key verbs: "conduct", "show", "risk", "maintain", "reveal", "ground", "carry", "insure"
Verb-object combinations: "show be", "risk accident", "maintain risk", "reveal be", "ground tanker", "insure oil"
Subject-verb combinations: "captain conduct", "ship risk", "lawyer carry", "company insure"
Generalizations: GROUP, PHYSICAL-EVENT, BOAT, PERSON, FUEL, ENTERPRISE
Contexts: LABOR, TRANSPORTATION, WATER-TRANSPORTATION, COMMERCE, LAW
Unknowns: "quarrel", "ground", "argo merchant", "contente", "negligent"
Unresolved ambiguities: "way", "conduct", "reveal", "maintain", "point"

Classifications on this document:

OIL-SPILL 2.35 0.194
DISASTER 0.6 0.034
LABOR 1.6 0.095

The following comment is pertinent mainly to the next section. Note, in the last example, that DISASTER and LABOR were mildly tagged (the second number is the degree of classification). Labor is probably an erroneous labeling due to the word "crew". At any rate, these tags are much less significant than OIL-SPILL. There were about 15 models loaded at the time the document was classified. The reason "contente" is misspelled is that it is not a known word to the system. In the process of removing the "-ion" suffix, it opted for the "-te" spelling. Note that this would not make a difference for most applications, as long as it is treated consistently.

3. The Model-based Document Tagger and Model Editor

The document tagger looks at the SELECTED-KEYWORDS output of the parsing, and, based on that, tries to select "tags" or classifications for the document. It can only select classifications for which a "model description" is loaded. In the case of the first sample in the above section, the document tagger used "oil," "ship," "leak," "oil tanker," and "argo merchant" as clues to determine that the correct classification of the document should be OIL-SPILL.

The document tagger is not a particularly complex system itself: it basically looks for a more-or-less exact match with the output of the keyword phase (described above). The more interesting part of this work is the model editor. As a matter of philosophy, it was deemed preferable to have a complex model and a simple tagger, rather than a simple model with a more complex tagger. Thus the model editor must work much harder to insure that a very general and complete model is created. The model editor contains commands for creating, loading, editing, printing, and dumping models.

The model editor accepts a model name from a user and steps through the different keyword categories (CLASSIFIERS, KEY-NOUNS, CONTEXTS, etc.). First it informs the user about any existing triggers, for example, that "labor relation" and "strike fund" are existing keywords in the model. Then it asks for new ones. It makes various syntactic checks--for example, classifiers must be at least two words. Then it asks for a scaled rating of the importance of the keyword--5 means a very good keyword, 1 very poor but still better than random chance, 0 means a negative weight. (These numbers may eventually represent different types of classification such as "supporting keyword," etc.)

For some categories of keywords, the model editor will make various analyses

and perhaps ask if the user wants to add other items too. For example, if the user gives "car" as a KEY-NOUN, it will ask if "vehicle" should also be added as a key noun. It does this by checking the dictionary "kind tree" or hierarchy. Generalizations of a key noun or verb are based on the word itself rather than on the word's meaning. Thus, the model editor may ask some very unusual questions about what to add as an additional keyword; for example, if the user specifies "car" as a keyword, the model editor may ask if "lisp function" should also be added as a key noun. The basic philosophy is to generalize and specialize at model-building time rather than at document-classification time. The trade-off is higher speed versus smaller size for the models. We believe we can easily keep a hundred models or so in primary storage.

When the user gives a keyword, it is looked up in the dictionary. If the word is not there, the model editor puts it into the unknown-word category. If the word is there, any CONTEXT indications in the dictionary definition are remembered, and the user is given a chance to add these automatically to the CONTEXT category of keyword at the appropriate time.

Currently we use about 15-20 models. It takes a person about two hours to compose a model using the editor. These models range from OIL-SPILL to JIMMY-CARTER to LABOR to SPORTS. We anticipate having about 100 models by the end of the summer (Dill). Many of the models will overlap--indeed some will be covering essentially the same subject, but at different levels. The system allows multiple authors to input their own versions of the same model topic and even to use the same model name.

The model builder and editors are currently receiving much work. One type of matching that has not been included is that of exactly matching a string specified in the model to any position in the input sentence. The main reason for not doing this is that we are much more interested in high-level matching techniques. On the other hand, there have been several examples where such a simple technique would have proved very beneficial as an additional method--often due to an inadequacy of the English parser to disambiguate a word or recognize a special construction. We feel the proper place for effort is at the high-level heuristics. Still, though, an exact-match feature would be desirable for special applications, such as an inventory clerk who would be looking for any document which refers to a part-number containing the sequence "AB123," for example. We have allowed a provision in the document tagger for this type of technique and could easily add it.

REFERENCES

Note: The form XXX.nn.nn denotes a Programming Technology Group document.

1. M.I.T. Laboratory for Computer Science. Progress Report XIII July 1975-July 1976. Cambridge, Massachusetts.

2. Church, Ken W. A System for Understanding Morse Networks. M.I.T., UROP Report, May 1976.

3. Anderson, Timothy A. "Modelling Morse-code Senders." S. M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

4. Eisenstadt, B. M.; Gold, B.; Nelson, D. M.; Pitcher, T. S.; and Selfridge, O. G. MAUDE (Morse Automatic Decoder). M.I.T., Lincoln Laboratory Group Report 34-57, December 1958.

5. Vezza, A. 1977 Proposal for Continuation of Research: Morse Code. M.I.T., Laboratory for Computer Science, SYS.53.02, expected date of completion, June 1977.

6. Van Trees, Harry L. Detection, Estimation, and Modulation Theory; Part II: Nonlinear Modulation Theory. New York: John Wiley and Sons, Inc., 1971.

7. Gardner, Floyd M. Phase Lock Techniques. New York: John Wiley and Sons, Inc., 1966.

8. Vezza, A., and Broos, M. S. "An Electronic Message System: Where Does it Fit?". Proceedings of IEEE Symposium, Trends and Applications 1976: Computer Networks. New York, New York, November 1976.

9. Galley, S. W. Data-based Message Service User's Manual. M.I.T., Laboratory for Computer Science, Programming Technology Group, expected date of publication, February 1977.

10. Martin, Bob; Oestreicher, Don; and Stotz, Rob. HP/MME Application Specification. University of Southern California, Information Sciences Institute, preliminary working paper. May 1976.

11. Galley, S. W. and Pfister, Greg. MDL Programming Language Primer and Manual. M.I.T., Laboratory for Computer Science, Cambridge, Massachusetts, expected date of publication, May 1977.

12. Myer, Theodore H., and Mooers, Charlotte D. Hermes Users' Guide (draft). Bolt, Beranek and Newman, Inc., Cambridge, Ma., June 1976.

13. Heafner, John F.; Miller, Lawrence H.; and Zogby, Bonnie Arter. Sigma Message Service Reference Manual. University of Southern California, Information Sciences Institute, Working Paper ISI/WP-5, to appear.

14. Martin, James. "Computer Data Base Organization". Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

15. Salton, Gerard, editor. The SMART Retrieval System: Experiments in Automatic Document Processing. Englewood Cliffs, New Jersey: Prentice-Hall, 1971.

Publications

1. Vezza, A., and Broos, M. S. "An Electronic Message System: Where Does it Fit?". Proceedings of IEEE Symposium, Trends and Applications 1976: Computer Networks. New York, New York, November 1976.

Talks

1. Vezza, A., and Broos, M.S. "An Electronic Message System: Where Does it Fit?" National Computer Conference, New York, June 1976.

Theses Completed

1. Cutler, Scott E. Microcomputer Networks in Control Applications. unpublished Ph.D. thesis, M.I.T., Department of Electrical Engineering and Computer Science, April 1976.

Theses in Progress

1. Anderson, T. A. "Modelling Morse-code Senders." unpublished M. S. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, May 1977.

2. Berez, Joel M. "A Dynamic Debugging System for MDL." unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1977.

To be Published

1. Galley, S.W. and Pfister, Greg. MDL Programming Language Primer and Manual. M.I.T., Laboratory for Computer Science, Cambridge, Massachusetts, expected date of publication, May 1977.

LABORATORY FOR COMPUTER SCIENCE

PUBLICATIONS

TECHNICAL MEMORANDA

TM-10 Jackson, James N.
Interactive Design Coordination for the Building Industry
June 1970
AD 708-400

*TM-11 Ward, Philip W.
Description and Flow Chart of the PDP-7/9 Communications Package
July 1970
AD 711-379

*TM-12 Graham, Robert M.
File Management and Related Topics (Formerly Programming Linguistics Group Memo No. 6, June 12, 1970)
September 1970
AD 712-068

*TM-13 Graham, Robert M.
Use of High Level Languages for Systems Programming (Formerly Programming Linguistics Group Memo No. 2, November 20, 1969)
September 1970
AD 711-965

*TM-14 Vogt, Carla M.
Suspension of Processes in a Multi-processing Computer System (Based on M.S. Thesis, EE Dept., February 1970)
September 1970
AD 713-989

TMs 1-9 were never issued

*TM-15 Zilles, Stephen N.
An Expansion of the Data Structuring Capabilities of PAL (Based on M.S. Thesis, EE Dept., June 1970)
October 1970

AD 720-761

*TM-16 Bruere-Dawson, Gerard
Pseudo-Random Sequences (Based on M.S. Thesis, EE Dept., June 1970)
October 1970

AD 713-852

*TM-17 Goodman, Leonard I.
Complexity Measures for Programming Languages (Based on M.S. Thesis, EE Dept., September 1971)
September 1971

AD 729-011

*TM-18 Reprinted as TR-85

*TM-19 Fenichel, Robert R.
A New List-Tracing Algorithm
October 1970

AD 714-522

*TM-20 Jones, Thomas L.
A Computer Model of Simple Forms of Learning (Based on Ph.D. Thesis, EE Dept., September 1970)
January 1971

AD 720-337

*TM-21 Goldstein, Robert, C.
The Substantive Use of Computers for Intellectual Activities
April 1971

AD 721-618

*TM-22 Wells, Douglas M.
Transmission of Information Between a Man-Machine Decision System and Its Environment
April 1971
AD 722-837

TM-23 Strnad, Alois J.
The Relational Approach to the Management of Data Bases
April 1971
AD 721-619

*TM-24 Goldstein, Robert C., and Alois J. Strnad
The MacAIMS Data Management System
April 1971
AD 721-620

TM-25 Goldstein, Robert C.
Helping People Think
April 1971
AD 721-998

TM-26 Iazeolla, Giuseppe G.
Modeling and Decomposition of Information Systems for Performance Evaluation
June 1971
AD 733-965

*TM-27 Bagchi, Amitava
Economy of Descriptions and Minimal Indices
January 1972
AD 736-960

TM-28 Wong, Richard
Construction Heuristics for Geometry and a Vector Algebra Representation of Geometry
June 1972
AD 743-487

*TM-29 Hossley, Robert and Charles Rackoff
The Emptiness Problem for Automata on Infinite Trees
Spring 1972

AD 747-250

*TM-30 McCray, William A.
SIM360: A S/360 Simulator
(Based on B.S. Thesis, ME Dept., May 1972)
October 1972

AD 749-365

TM-31 Bonneau, Richard J.
A Class of Finite Computation Structures Supporting the Fast Fourier Transform
March 1973

AD 757-787

TM-32 Moll, Robert
An Operator Embedding Theorem for Complexity Classes of Recursive Functions
May 1973

AD 759-999

*TM-33 Ferrante, Jeanne and Charles Rackoff
A Decision Procedure for the First Order Theory of Real Addition with Order
May 1973

AD 760-000

*TM-34 Bonneau, Richard J.
Polynomial Exponentiation: The Fast Fourier Transform Revisited
June 1973

PB 221-742

TM-35 Bonneau, Richard J.
An Interactive Implementation of the Todd-Coxeter Algorithm
December 1973

AD 770-565

TM-36 Geiger, Steven P.
A User's Guide to the Macro Control Language
December 1973
AD 771-435

*TM-37 Schoenhage, A.
Real-Time Simulation of Multidimensional Turing Machines by Storage Modification Machines
December 1973
PB 226-103/AS

*TM-38 Meyer, Albert R.
Weak Monadic Second Order Theory of Succesor is not Elementary-Recursive
December 1973
PB 226-514/AS

TM-39 Meyer, Albert R.
Discrete Computation: Theory and Open Problems
January 1974
PB 226-836/AS

TM-40 Paterson, Michael S., Michael J. Fischer and Albert R. Meyer
An Improved Overlap Argument for On-Line Multiplication
January 1974
AD 773-137

TM-41 Fischer, Michael J., and Michael S. Paterson
String-Matching and Other Products
January 1974
AD 773-138

*TM-42 Rackoff, Charles
On the Complexity of the Theories of Weak Direct Products
January 1974
PB 228-459/AS

TM-43 Fischer, Michael J., and Michael O. Rabin
Super-Exponential Complexity of Presburger Arithmetic
February 1974
AD 775-004

TM-44 Pless, Vera
Symmetry Codes and their Invariant Subcodes
May 1974
AD 780-243

*TM-45 Fischer, Michael J., and Larry J. Stockmeyer
Fast On-Line Integer Multiplication
May 1974
AD 779-889

*TM-46 Kedem, Zvi M.
Combining Dimensionality and Rate of Growth Arguments for Establishing Lower Bounds on the Number of Multiplications
June 1974
PB 232-969/AS

TM-47 Pless, Vera
Mathematical Foundations of Flip-Flops
June 1974
AD 780-901

TM-48 Kedem, Zvi M.
The Reduction Method for Establishing Lower Bounds on the Number of Additions
June 1974
PB 233-538/AS

TM-49 Pless, Vera
Complete Classification of (24,12) and (22,11) Self-Dual Codes
June 1974
AD 781-335

TM-50 Benedict, G. Gordon
An Enciphering Module for Multics
B.S. Thesis, EE Dept.
July 1974

AD 782-658

*TM-51 Aiello, Jack M.
An Investigation of Current Language Support for the Data Requirements of Structured Programming
M.S. & E.E. Theses, EE Dept.
September 1974

PB 236-815/AS

TM-52 Lind, John C.
Computing in Logarithmic Space
September 1974

PB 236-167/AS

TM-53 Bengelloun, Safwan A.
MDC-Programmer: A Muddle-to Datalanguage Translator for Information Retrieval
B.S. Thesis, EE Dept.
October 1974

AD 786-754

*TM-54 Meyer, Albert. R.
The Inherent Computation Complexity of Theories of Ordered Sets: A Brief Survey
October 1974

PB 237-200/AS

TM-55 Hsieh, Wen N., Larry H. Harper and John E. Savage
A Class of Boolean Functions with Linear Combinatorial Complexity
October 1974

PB 237-206/AS

TM-56 Gorry, G. Anthony
Research on Expert Systems
December 1974

TM-57 Levin, Michael
On Bateson's Logical Levels of Learning
February 1975

TM-58 Qualitz, Joseph E.
Decidability of Equivalence for a Class of Data Flow Schemas
March 1975

PB 237-033/AS

*TM-59 Hack, Michel
Decision Problems for Petri Nets and Vector Addition Systems
March 1975

PB 231-916/AS

TM-60 Weiss, Randell B.
CAMAC: Group Manipulation System
March 1975

PB 240-495/AS

TM-61 Dennis, Jack B.
First Version of a Data Flow Procedure Language
May 1975

TM-62 Patil, Suhas S.
An Asynchronous Logic Array
May 1975

TM-63 Pless, Vera
Encryption Schemes for Computer Confidentiality
May 1975

AD A010-217

*TM-64 Weiss, Randell B.
Finding Isomorph Classes for Combinatorial Structures
M.S. Thesis, EE Dept.
June 1975

TM-65 Fischer, Michael J.
The Complexity Negation-Limited Networks - A Brief Survey
June 1975

*TM-66 Leung, Clement
Formal Properties of Well-Formed Data Flow Schemas
B.S., M.S. & E.E. Theses, EE Dept.
June 1975

*TM-67 Cardoza, Edward E.
Computational Complexity of the Word Problem for Commutative Semigroups
M.S. Thesis, EE & CS Dept.
October 1975

TM-68 Weng, Kung-Song
Stream-Oriented Computation in Recursive Data Flow Schemas
M.S. Thesis, EE & CS Dept.
October 1975

*TM-69 Bayer, Paul J.
Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees
M.S. Thesis, EE & CS Dept.
November 1975

TM-70 Ruth, Gregory R.
Automatic Design of Data Processing Systems
February 1976

AD A023-451

*TM-71 Rivest, Ronald
On the Worst-Case of Behavior of String-Searching Algorithms
April 1976

*TM-72 Ruth, Gregory R.
Protosystem I: An Automatic Programming System Prototype
July 1976

AD A026-912

TM-73 Rivest, Ronald
Optimal Arrangement of Keys in a Hash Table
July 1976

TM-74 Malvania, Nikhil
The Design of a Modular Laboratory for Control Robotics
M.S. Thesis, EE & CS Dept.
September 1976

AD A030-418

TM-75 Yao, Andrew C., and Ronald I. Rivest
K+1 Heads are Better than K
September 1976

AD A030-008

*TM-76 Bloniaz, Peter A., Michael J. Fischer and Albert R. Meyer
A Note on the Average Time to Compute Transitive Closures
September 1976

TM-77 Mok, Aloysius K.
Task Scheduling in the Control Robotics Environment
M.S. Thesis, EE & CS Dept.
September 1976

AD A030-402

*TM-78 Benjamin, Arthur J.
Improving Information Storage Reliability
Using a Data Network
M.S. Thesis, EE & CS Dept.
October 1976

AD A033-394

TM-79 Brown, Gretchen P.
A System to Process Dialogue: A Progress Report
October 1976

AD A033-276

TM-80 Even, Shimon
The Max Flow Algorithm of Dinic and Karzanov:
An Exposition
December 1976

TECHNICAL REPORTS

*TR-1 Bobrow, Daniel G.
Natural Language Input for a Computer Problem Solving System,
Ph.D. Thesis, Math. Dept.
September 1964

AD 604-730

*TR-2 Raphael, Bertram
SIR: A Computer Program for Semantic Information Retrieval,
Ph.D. Thesis, Math. Dept.
June 1964

AD 608-499

*TR-3 Corbato, Fernando J.
System Requirements for Multiple-Access, Time-Shared Computers
May 1964

AD 608-501

*TR-4 Ross, Douglas T., and Clarence G. Feldman
Verbal and Graphical Language for the AED System: A Progress Report
May 1964

AD 604-678

*TR-6 Biggs, John M., and Robert D. Logcher
STRESS: A Problem-Oriented Language for Structural Engineering
May 1964

AD 604-679

TRs 5, 9, 10, 15 were never issued

*TR-7 Weizenbaum, Joseph
OPL-1: An Open Ended Programming System within CTSS
April 1964
AD 604-680

*TR-8 Greenberger, Martin
The OPS-1 Manual
May 1964
AD 604-681

*TR-11 Dennis, Jack B.
Program Structure in a Multi-Access Computer
May 1964
AD 608-500

*TR-12 Fano, Robert M.
The MAC System: A Progress Report
October 1964
AD 609-296

*TR-13 Greenberger, Martin
A New Methodology for Computer Simulation
October 1964
AD 609-288

*TR-14 Roos, Daniel
Use of CTSS in a Teaching Environment
November 1964
AD 661-807

*TR-16 Saltzer, Jerome H.
CTSS Technical Notes
March 1965
AD 612-702

*TR-17 Samuel, Arthur L.
Time-Sharing on a Multiconsole Computer
March 1965
AD 462-158

*TR-18 Scherr, Allan Lee
An Analysis of Time-Shared Computer Systems,
Ph.D. Thesis, EE Dept.
June 1965

AD 470-715

*TR-19 Russo, Francis John
A Heuristic Approach to Alternate Routing in a Job Shop,
B.S. & M.S. Theses, Sloan School
June 1965

AD 474-018

*TR-20 Wantman, Mayer Elihu
CALCULAID: An On-Line System for
Algebraic Computation and Analysis,
M.S. Thesis, Sloan School
September 1965

AD 474-019

*TR-21 Denning, Peter James
Queueing Models for File Memory Operation,
M.S. Thesis, EE Dept.
October 1965

AD 624-943

*TR-22 Greenberger, Martin
The Priority Problem
November 1965

AD 625-728

*TR-23 Dennis, Jack B., and Earl C. Van Horn
Programming Semantics for Multi-
programmed Computations
December 1965

AD 627-537

*TR-24 Kaplow, Roy, Stephen Strong and John Brackett
MAP: A System for On-Line Mathematical
Analysis
January 1966

AD 476-443

*TR-25 Stratton, William David
Investigation of an Analog Technique
to Decrease Pen-Tracking Time in
Computer Displays,
M.S. Thesis, EE Dept.
March 1966

AD 631-396

*TR-26 Cheek, Thomas Burrell
Design of a Low-Cost Character
Generator for Remote Computer Displays,
M.S. Thesis, EE Dept.
March 1966

AD 631-269

*TR-27 Edwards, Daniel James
OCAS - On-Line Cryptanalytic Aid
System,
M.S. Thesis, EE Dept.
May 1966

AD 633-678

*TR-28 Smith, Arthur Anshel
Input/Output in Time-Shared, Segmented,
Multiprocessor Systems,
M.S. Thesis, EE Dept.
June 1966

AD 637-215

*TR-29 Ivie, Evan Leon
Search Procedures Based on Measures
of Relatedness between Documents,
Ph.D. Thesis, EE Dept.
June 1966

AD 636-275

TR-30 Saltzer, Jerome Howard
Traffic Control in a Multiplexed
Computer System,
Sc.D. Thesis, EE Dept.
July 1966

AD 635-966

*TR-31 Smith, Donald L.
Models and Data Structures for Digital Logic Simulation,
M.S. Thesis, EE Dept.
August 1966

AD 637-192

*TR-32 Teitelman, Warren
PILOT: A Step Toward Man-Computer Symbiosis,
Ph.D. Thesis, Math. Dept.
September 1966

AD 638-446

*TR-33 Norton, Lewis M.
ADEPT - A Heuristic Program for Proving Theorems of Group Theory,
Ph.D. Thesis, Math. Dept.
October 1966

AD 645-660

*TR-34 Van Horn, Earl C., Jr.
Computer Design for Asynchronously Reproducible Multiprocessing,
Ph.D. Thesis, EE Dept.
November 1966

AD 650-407

*TR-35 Fenichel, Robert R.
An On-Line System for Algebraic Manipulation,
Ph.D. Thesis, Appl. Math. (Harvard)
December 1966

AD 657-282

*TR-36 Martin, William A.
Symbolic Mathematical Laboratory,
Ph.D. Thesis, EE Dept.
January 1967

AD 657-283

*TR-37 Guzman-Arenas, Adolfo
Some Aspects of Pattern Recognition by Computer,
M.S. Thesis, EE Dept.
February 1967

AD 656-041

*TR-38 Rosenberg, Ronald C., Daniel W. Kennedy and Roger A. Humphrey
A Low-Cost Output Terminal For Time-Shared Computers
March 1967

AD 662-027

*TR-39 Forte, Allen
Syntax-Based Analytic Reading of Musical Scores
April 1967

AD 661-806

*TR-40 Miller, James R.
On-Line Analysis for Social Scientists
May 1967

AD 668-009

*TR-41 Coons, Steven A.
Surfaces for Computer-Aided Design of Space Forms
June 1967

AD 663-504

*TR-42 Liu, Chung L., Gabriel D. Chang and Richard E. Marks
Design and Implementation of a Table-Driven Compiler System
July 1967

AD 668-960

*TR-43 Wilde, Daniel U.
Program Analysis by Digital Computer,
Ph.D. Thesis, EE Dept.
August 1967

AD 662-224

*TR-44 Gorry, G. Anthony
A System for Computer-Aided Diagnosis,
Ph.D. Thesis, Sloan School
September 1967

AD 662-665

*TR-45 Leal-Cantu, Nestor
On the Simulation of Dynamic Systems
with Lumped Parameters and Time Delays,
M.S. Thesis, ME Dept.
October 1967

AD 663-502

*TR-46 Alsop, Joseph W.
A Canonic Translator,
B.S. Thesis, EE Dept.
November 1967

AD 663-503

*TR-47 Moses, Joel
Symbolic Integration,
Ph.D. Thesis, Math. Dept.
December 1967

AD 662-666

*TR-48 Jones, Malcolm M.
Incremental Simulation on a Time-
Shared Computer,
Ph.D. Thesis, Sloan School
January 1968

AD 662-225

*TR-49 Luconi, Fred L.
Asynchronous Computational Structures,
Ph.D Thesis, EE Dept.
February 1968

AD 667-602

*TR-50 Denning, Peter J.
Resource Allocation in Multiprocess
Computer Systems,
Ph.D. Thesis, EE Dept.
May 1968

AD 675-554

*TR-51 Charniak, Eugene
CARPS, A Program which Solves Calculus Word Problems,
M.S. Thesis, EE Dept.
July 1968

AD 673-670

*TR-52 Deitel, Harvey M.
Absentee Computations in a Multiple-Access Computer System,
M.S. Thesis, EE Dept.
August 1968

AD 684-738

*TR-53 Slutz, Donald R.
The Flow Graph Schemata Model of Parallel Computation,
Ph.D. Thesis, EE Dept.
September 1968

AD 683-393

*TR-54 Grochow, Jerrold M.
The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System,
M.S. Thesis, EE Dept.
October 1968

AD 689-468

*TR-55 Rappaport, Robert L.
Implementing Multi-Process Primitives in a Multiplexed Computer System,
M.S. Thesis, EE Dept.
November 1968

AD 689-469

*TR-56 Thornhill, Daniel E., Robert H. Stotz, Douglas T. Ross and John E. Ward (ESL-R-356)
An Integrated Hardware-Software System for Computer Graphics in Time-Sharing
December 1968

AD 685-202

*TR-57 Morris, James H.
Lambda-Calculus Models of Programming Languages,
Ph.D. Thesis, Sloan School
December 1968

AD 683-394

*TR-58 Greenbaum, Howard J.
A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System,
M.S. Thesis, EE Dept.
January 1969

AD 686-988

*TR-59 Guzman, Adolfo
Computer Recognition of Three-Dimensional Objects in a Visual Scene,
Ph.D. Thesis, EE Dept.
December 1968

AD 692-200

*TR-60 Ledgard, Henry F.
A Formal System for Defining the Syntax and Semantics of Computer Languages,
Ph.D. Thesis, EE Dept.
April 1969

AD 689-305

*TR-61 Baecker, Ronald M.
Interactive Computer-Mediated Animation,
Ph.D. Thesis, EE Dept.
June 1969

AD 690-887

*TR-62 Tillman, Coyt C., Jr. (ESL-R-395)
EPS: An Interactive System for Solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation
June 1969

AD 692-462

*TR-63 Brackett, John W., Michael Hammer and Daniel E. Thornhill
Case Study in Interactive Graphics Programming: A Circuit Drawing and Editing Program for Use with a Storage-Tube Display Terminal
October 1969

AD 699-930

*TR-64 Rodriguez, Jorge E. (ESL-R-398)
A Graph Model for Parallel Computations,
Sc.D. Thesis, EE Dept.
September 1969

AD 697-759

*TR-65 DeRemer, Franklin L.
Practical Translators for LR(k) Languages,
Ph.D. Thesis, EE Dept.
October 1969

AD 699-501

*TR-66 Beyer, Wendell T.
Recognition of Topological Invariants by Iterative Arrays,
Ph.D. Thesis, Math. Dept.
October 1969

AD 699-502

*TR-67 Vanderbilt, Dean H.
Controlled Information Sharing in a Computer Utility,
Ph.D. Thesis, EE Dept.
October 1969

AD 699-503

*TR-68 Selwyn, Lee L.
Economies of Scale in Computer Use: Initial Tests and Implications for The Computer Utility,
Ph.D. Thesis, Sloan School
June 1970

AD 710-011

*TR-69 Gertz, Jeffrey L.
Hierarchical Associative Memories for Parallel Computation,
Ph.D. Thesis, EE Dept.
June 1970

AD 711-091

*TR-70 Fillat, Andrew I., and Leslie A. Kraning
Generalized Organization of Large Data-Bases: A Set-Theoretic Approach to Relations,
B.S. & M.S. Theses, EE Dept.
June 1970

AD 711-060

*TR-71 Fiasconaro, James G.
A Computer-Controlled Graphical Display Processor,
M.S. Thesis, EE Dept.
June 1970

AD 710-479

TR-72 Patil, Suhas S.
Coordination of Asynchronous Events,
Sc.D. Thesis, EE Dept.
June 1970

AD 711-763

*TR-73 Griffith, Arnold K.
Computer Recognition of Prismatic Solids,
Ph.D. Thesis, Math. Dept.
August 1970

AD 712-069

TR-74 Edelberg, Murray
Integral Convex Polyhedra and an Approach to Integralization,
Ph.D. Thesis, EE Dept.
August 1970

AD 712-070

*TR-75 Hebalkar, Prakash G.
Deadlock-Free Sharing of Resources
in Asynchronous Systems,
Sc.D. Thesis, EE Dept.
September 1970

AD 713-139

*TR-76 Winston, Patrick H.
Learning Structural Descriptions
from Examples,
Ph.D. Thesis, EE Dept.
September 1970

AD 713-988

TR-77 Haggerty, Joseph P.
Complexity Measures for Language
Recognition by Canonic Systems,
M.S. Thesis, EE Dept.
October 1970

AD 715-134

*TR-78 Madnick, Stuart E.
Design Strategies for File Systems,
M.S. Thesis, EE Dept. & Sloan School
October 1970

AD 714-269

TR-79 Horn, Berthold K.
Shape from Shading: A Method for
Obtaining the Shape of a Smooth
Opaque Object from One View,
Ph.D. Thesis, EE Dept.
November 1970

AD 717-336

TR-80 Clark, David D., Robert M. Graham,
Jerome H. Saltzer and Michael D. Schroeder
The Classroom Information and Computing
Service
January 1971

AD 717-857

TR-81 Banks, Edwin R.
Information Processing and Transmission in Cellular Automata,
Ph.D. Thesis, ME Dept.
January 1971
AD 717-951

*TR-82 Krakauer, Lawrence J.
Computer Analysis of Visual Properties of Curved Objects,
Ph.D. Thesis, EE Dept.
May 1971
AD 723-647

*TR-83 Lewin, Donald E.
In-Process Manufacturing Quality Control,
Ph.D. Thesis, Sloan School
January 1971
AD 720-098

*TR-84 Winograd, Terry
Procedures as a Representation for Data in a Computer Program for Understanding Natural Language,
Ph.D. Thesis, Math. Dept.
February 1971
AD 721-399

TR-85 Miller, Perry L.
Automatic Creation of a Code Generator from a Machine Description,
E.E. Thesis, EE Dept.
May 1971
AD 724-730

*TR-86 Schell, Roger R.
Dynamic Reconfiguration in a Modular Computer System,
Ph.D. Thesis, EE Dept.
June 1971
AD 725-859

TR-87 Thomas, Robert H.
A Model for Process Representation and Synthesis,
Ph.D. Thesis, EE Dept.
June 1971
AD 726-049

TR-88 Welch, Terry A.
Bounds on Information Retrieval Efficiency in Static File Structures,
Ph.D. Thesis, EE Dept.
June 1971
AD 725-429

TR-89 Owens, Richard C., Jr.
Primary Access Control in Large-Scale Time-Shared Decision Systems,
M.S. Thesis, Sloan School
July 1971
AD 728-036

TR-90 Lester, Bruce P.
Cost Analysis of Debugging Systems,
B.S. & M.S. Theses, EE Dept.
September 1971
AD 730-521

*TR-91 Smoliar, Stephen W.
A Parallel Processing Model of Musical Structures,
Ph.D. Thesis, Math. Dept.
September 1971
AD 731-690

TR-92 Wang, Paul S.
Evaluation of Definite Integrals by Symbolic Manipulation
Ph.D. Thesis, Math. Dept.
October 1971
AD 732-005

TR-93 Greif, Irene Gloria
Induction in Proofs about Programs,
M.S. Thesis, EE Dept.
February 1972
AD 737-701

TR-94 Hack, Michel Henri Theodore
Analysis of Production Schemata
by Petri Nets,
M.S. Thesis, EE Dept.
February 1972
AD 740-320

TR-95 Fateman, Richard J.
Essays in Algebraic Simplification
(A revision of a Harvard Ph.D. Thesis)
April 1972
AD 740-132

TR-96 Manning, Frank
Autonomous, Synchronous Counters Constructed Only of
J-K Flip-Flops,
M.S. Thesis, EE Dept.
May 1972
AD 744-030

TR-97 Vilfan, Bostjan
The Complexity of Finite Functions
Ph.D. Thesis, EE Dept.
March 1972
AD 739-678

TR-98 Stockmeyer, Larry Joseph
Bounds on Polynomial Evaluation Algorithms
M.S. Thesis, EE Dept.
April 1972
AD 740-328

TR-99 Lynch, Nancy Ann
Relativization of the Theory of Computational Complexity
Ph.D. Thesis, Math. Dept.
June 1972
AD 744-032

TR-100 Mandl, Robert
Further Results on Hierarchies of Canonic Systems
M.S. Thesis, EE Dept.
June 1972

AD 744-206

TR-101 Dennis, Jack B.
On the Design and Specification of a Common Base Language
June 1972

AD 744-207

TR-102 Hossley, Robert F.
Finite Tree Automata and ω-Automata
M.S. Thesis, EE Dept.
September 1972

AD 749-367

*TR-103 Sekino, Akira
Performance Evaluation of Multiprogrammed Time-Shared Computer Systems
Ph.D Thesis, EE Dept.
September 1972

AD 749-949

TR-104 Schroeder, Michael D.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility
Ph.D. Thesis, EE Dept.
September 1972

AD 750-173

TR-105 Smith, Burton J.
An Analysis of Sorting Networks
Sc.D. Thesis, EE Dept.
October 1972

AD 751-614

TR-106 Rackoff, Charles W.
The Emptiness and Complementation Problems for Automata on Infinite Trees
M.S. Thesis, EE Dept.
January 1973

AD 756-248

TR-107 Madnick, Stuart E.
Storage Hierarchy Systems
Ph.D. Thesis, EE Dept.
April 1973

AD 760-001

TR-108 Wand, Mitchell
Mathematical Foundations of Formal Language Theory
Ph.D. Thesis, Math. Dept.
December 1973

TR-109 Johnson, David S.
Near-Optimal Bin Packing Algorithms
Ph.D. Thesis, Math. Dept.
June 1973

PB 222-090

TR-110 Moll, Robert
Complexity Classes of Recursive Functions
Ph.D. Thesis, Math. Dept.
June 1973

AD 767-730

TR-111 Linderman, John P.
Productivity in Parallel Computation Schemata
Ph.D. Thesis, EE Dept.
December 1973

PB 226-159/AS

TR-112 Hawryszkiewycz, Igor T.
Semantics of Data Base Systems
Ph.D. Thesis, EE Dept.
December 1973

PB 226-061/AS

TR-113 Herrmann, Paul P.
On Reducibility Among Combinatorial Problems
M.S. Thesis, Math. Dept.
December 1973

PB 226-157/AS

TR-114 Metcalfe, Robert M.
Packet Communication
Pn.D. Thesis, Applied Math., Harvard University
December 1973

AD 771-430

TR-115 Rotenberg, Leo
Making Computers Keep Secrets
Ph.D Thesis, EE Dept.
February 1974

PB 229-352/AS

TR-116 Stern, Jerry A.
Backup and Recovery of On-Line Information
in a Computer Utility
M.S. & E.E. Theses, EE Dept.
January 1974

AD 774-141

TR-117 Clark, David D.
An Input/Output Architecture for
Virtual Memory Computer Systems
Ph.D. Thesis, EE Dept.
January 1974

AD 774-738

TR-118 Briabrin, Victor
An Abstract Model of a Research Institute:
Simple Automatic Programming Approach
March 1974

PB 231-505/AS

TR-119 Hammer, Michael M.
A New Grammatical Transformation into
Deterministic Top-Down Form
Ph.D. Thesis, EE Dept.
February 1974

AD 775-545

TR-120 Ramchandani, Chander
Analysis of Asynchronous Concurrent Systems
by Timed Petri Nets
Ph.D. Thesis, EE Dept.
February 1974

AD 775-618

TR-121 Yao, Foong F.
On Lower Bounds for Selection Problems
Ph.D. Thesis, Math. Dept.
March 1974

PB 230-950/AS

TR-122 Scherf, John A.
Computer and Data Security: A Comprehensive Annotated Bibliography
M.S. Thesis, Sloan School
January 1974

AD 775-546

TR-123 Introduction to Multics
February 1974

AD 918-562

TR-124 Laventhal, Mark S.
Verification of Programs Operating on Structured Data
B.S. & M.S. Theses, EE Dept.
March 1974

PB 231-365/AS

TR-125 Mark, William S.
A Model-Debugging System
B.S. & M.S. Theses, EE Dept.
April 1974

AD 778-688

TR-126 Altman, Vernon E.
A Language Implementation System
B.S. & M.S. Theses, Sloan School
May 1974

AD 780-672

TR-127 Greenberg, Bernard S.
An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory
M.S. Thesis, EE Dept.
May 1974

AD 780-407

TR-128 Frankston, Robert M.
The Computer Utility as a Marketplace for Computer Services
M.S. & E.E. Theses, EE Dept.
May 1974
AD 780-436

TR-129 Welssberg, Richard W.
Using Interactive Graphics in Simulating the Hospital Emergency Room
M.S. Thesis, EE Dept.
May 1974
AD 780-437

TR-130 Ruth, Gregory R.
Analysis of Algorithm Implementations
Ph.D. Thesis, EE Dept.
May 1974
AD 780-408

TR-131 Levin, Michael
Mathematical Logic for Computer Scientists
June 1974

TR-132 Janson, Philippe A.
Removing the Dynamic Linker from the Security Kernel of a Computing Utility
M.S. Thesis, EE Dept.
June 1974
AD 781-305

TR-133 Stockmeyer, Larry J.
The Complexity of Decision Problems in Automata Theory and Logic
Ph.D. Thesis, EE Dept.
July 1974
PB 235-283/AS

TR-134 Ellis, David J.
Semantics of Data Structures and References
M.S. & E.E. Theses, EE Dept.
August 1974
PB 236-594/AS

TR-135 Pfister, Gregory F.
The Computer Control of Changing Pictures
Ph.D. Thesis, EE Dept.
September 1974

AD 787-795

TR-136 Ward, Stephen A.
Functional Domains of Applicative Languages
Ph.D. Thesis, EE Dept.
September 1974

AD 787-796

TR-137 Seiferas, Joel I.
Nondeterministic Time and Space Complexity Classes
Ph.D Thesis, Math. Dept.
September 1974

PB 236-777/AS

TR-138 Yun, David Y. Y.
The Hensel Lemma in Algebraic Manipulation
Ph.D. Thesis, Math. Dept.
November 1974

AD A002-737

TR-139 Ferrante, Jeanne
Some Upper and Lower Bounds on Decision Procedures in Logic
Ph.D. Thesis, Math. Dept.
November 1974

PB 238-121/AS

TR-140 Redell, David D.
Naming and Protection in Extendible Operating Systems
Ph.D. Thesis, EE Dept.
November 1974

AD A001-721

TR-141 Richards, Martin, A. Evans and R. Mabee
The BCPL Reference Manual
December 1974

AD A003-599

TR-142 Brown, Gretchen P.
Some Problems in German to English Machine Translation
M.S. & E.E. Theses, EE Dept.
December 1974
AD A003-002

TR-143 Silverman, Howard
A Digitalis Therapy Advisor
M.S. Thesis, EE Dept.
January 1975

TR-144 Rackoff, Charles
The Computational Complexity of Some Logical Theories
Ph.D. Thesis, EE Dept.
February 1975

*TR-145 Henderson, D. Austin
The Binding Model: A Semantic Base for Modular Programming Systems
Ph.D. Thesis, EE Dept.
February 1975
AD A006-961

*TR-146 Malhotra, Ashok
Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis
Ph.D. Thesis, EE Dept.
February 1975

TR-147 Van De Vanter, Michael L.
A Formalization and Correctness Proof of the CGOL Language System
M.S. Thesis, EE Dept.
March 1975

TR-148 Johnson, Jerry
Program Restructuring for Virtual Memory Systems
Ph.D. Thesis, EE Dept.
March 1975
AD A009-218

*TR-149 Snyder, Alan
A Portable Compiler for the Language C
B.S. & M.S. Theses, EE Dept.
May 1975

AD A010-218

*TR-150 Rumbaugh, James E.
A Parallel Asynchronous Computer Architecture for Data Flow Programs
Ph.D. Thesis, EE Dept.
May 1975

AD A010-918

TR-151 Manning, Frank B.
Automatic Test, Configuration, and Repair of Cellular Arrays
Ph.D. Thesis, EE Dept.
June 1975

AD A012-822

TR-152 Qualitz, Joseph E.
Equivalence Problems for Monadic Schemas
Ph.D. Thesis, EE Dept.
June 1975

AD A012-823

TR-153 Miller, Peter B.
Strategy Selection in Medical Diagnosis
M.S. Thesis, EE & CS Dept.
September 1975

TR-154 Greif, Irene
Semantics of Communicating Parallel Processes
Ph.D. Thesis, EE & CS Dept.
September 1975

AD A016-302

TR-155 Kahn, Kenneth M.
Mechanization of Temporal Knowledge
M.S. Thesis, EE & CS Dept.
September 1975

TR-156 Bratt, Richard G.
Minimizing the Naming Facilities Requiring Protection in a Computer Utility
M.S. Thesis, EE & CS Dept.
September 1975

*TR-157 Meldman, Jeffrey A.
A Preliminary Study in Computer-Aided Legal Analysis
Ph.D. Thesis, EE & CS Dept.
November 1975

AD A018-997

TR-158 Grossman, Richard W.
Some Data-base Applications of Constraint Expressions
M.S. Thesis, EE & CS Dept.
February 1976

AD A024-149

TR-159 Hack, Michel
Petri Net Languages
March 1976

TR-160 Bosyj, Michael
A Program for the Design of Procurement Systems
M.S. Thesis, EE & CS Dept.
May 1976

AD A026-688

TR-161 Hack, Michel
Decidability Questions
Ph.D. Thesis, EE & CS Dept.
June 1976

TR-162 Kent, Stephen T.
Encryption-Based Protection Protocols for Interactive User-Computer Communication
M.S. Thesis, EE & CS Dept.
June 1976

AD A026-911

TR-163 Montgomery, Warren A.
A Secure and Flexible Model of Process Initiation for a Computer Utility
M.S. & E.E. Theses, EE & CS Dept.
June 1976

TR-164 Reed, David P.
Processor Multiplexing in a Layered Operating System
M.S. Thesis, EE & CS Dept.
July 1976

TR-165 McLeod, Dennis J.
High Level Expression of Semantic Integrity Specifications in a Relational Data Base System
M.S. Thesis, EE & CS Dept.
September 1976
AD A034-184

TR-166 Chan, Arvola Y.
Index Selection in a Self-Adaptive Relational Data Base Management System
M.S. Thesis, EE & CS Dept.
September 1976
AD A034-185

TR-167 Janson, Philippe A.
Using Type Extension to Organize Virtual Memory Mechanisms
Ph.D. Thesis, EE & CS Dept.
September 1976

TR-168 Pratt, Vaughan R.
Semantical Considerations on Floyd-Hoare Logic
September 1976

TR-169 Safran, Charles, James F. Desforges and Philip N. Tsichlis
Diagnostic Planning and Cancer Management
September 1976

TR-170 Furtek, Frederick C.
The Logic of Systems
Ph.D. Thesis, EE & CS Dept.
December 1976

TR-171 Huber, Andrew R.
A Multi-Process Design of a Paging System
M.S. & E.E. Theses, EE & CS Dept.
December 1976

TR-172 Mark, William S.
The Reformulation Model of Expertise
Ph.D. Thesis, EE & CS Dept.
December 1976

AD A035-397

TR-173 Goodman, Nathan
Coordination of Parallel Processes in the Actor Model of Computation
M.S. Thesis, EE & CS Dept.
December 1976

TR-174 Hunt, Douglas H.
A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem
M.S. & E.E. Theses, EE & CS Dept.
December 1976

PROGRESS REPORTS

*Project MAC Progress Report I
to July 1964

AD 465-088

*Project MAC Progress Report II
July 1964-July 1965

AD 629-494

*Project MAC Progress Report III
July 1965-July 1966

AD 648-346

*Project MAC Progress Report IV
July 1966-July 1967

AD 681-342

*Project MAC Progress Report V
July 1967-July 1968

AD 687-770

*Project MAC Progress Report VI
July 1968-July 1969

AD 705-434

*Project MAC Progress Report VII
July 1969-July 1970

AD 732-767

*Project MAC Progress Report VIII
July 1970-July 1971

AD 735-148

*Project MAC Progress Report IX
July 1971-July 1972

AD 756-689

*Project MAC Progress Report X
July 1972-July 1973

AD 771-428

*Project MAC Progress Report XI
July 1973-July 1974

AD A004-966

*Laboratory for Computer Science Progress Report XII
July 1974-July 1975

AD A024-527

*Laboratory for Computer Science Progress Report XIII
July 1975-July 1976

Copies of all reports with AD and PB numbers listed in Publications may be secured from the National Technical Information Service, Operations Division, Springfield, Virginia, 22151. Prices vary. The AD or PB number must be supplied with the request.

* Out of Print reports may be obtained from NTIS if the AD number is supplied (see above). Out of Print reports without an AD or PB number are unobtainable.

OFFICIAL DISTRIBUTION LIST

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Electronics Lab Center
Advanced Software Technology
Division - Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Captain Richard L. Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York, N. Y. 09501
1 copy